Mississippi State University
Scholars Junction

Theses and Dissertations | Theses and Dissertations

12-13-2003

Generalizing List Scheduling for Stochastic Soft Real-time Parallel Applications

Yoginder Singh Dandass

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
Dandass, Yoginder Singh, "Generalizing List Scheduling for Stochastic Soft Real-time Parallel Applications" (2003). *Theses and Dissertations*. 2386.
https://scholarsjunction.msstate.edu/td/2386

This Dissertation - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

GENERALIZING LIST SCHEDULING FOR STOCHASTIC SOFT REAL-TIME PARALLEL APPLICATIONS

By

Yoginder Singh Dandass

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State University

December 2003

Copyright © by

Yoginder Singh Dandass

2003

GENERALIZING LIST SCHEDULING FOR STOCHASTIC SOFT REAL-TIME PARALLEL APPLICATIONS

By

Yoginder Singh Dandass

Approved:

Anthony Skjellum Professor of Computer and Information Sciences, University of Alabama at Birmingham (Director of Dissertation)	Susan M. Bridges Professor of Computer Science and Engineering Committee Member and Graduate Coordinator in the Department of Computer Science and Engineering
Eric A. Hansen Associate Professor of Computer Science and Engineering (Committee Member)	Arkady Kanevsky Adjunct Faculty of Computer Science and Engineering (Committee Member)
Raghu Machiraju Adjunct Faculty of Computer Science and Engineering (Committee Member)	Donna S. Reese Professor of Computer Science and Engineering (Committee Member)

A. Wayne Bennett
Dean of the Bagley College of Engineering

Name: Yoginder Singh Dandass

Date of Degree: December 13, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: Generalizing List Scheduling for Stochastic Soft Real-Time Parallel Applications

Pages in Study: 247

Candidate for Degree of Doctor of Computer Science

Advanced architecture processors provide features such as caches and branch prediction that result in improved, but variable, execution time of software. Hard real-time systems require tasks to complete within timing constraints. Consequently, hard real-time systems are typically designed conservatively through the use of tasks' worst-case execution times (WCET) in order to compute deterministic schedules that guarantee task's execution within giving time constraints. This use of pessimistic execution time assumptions provides real-time guarantees at the cost of decreased performance and resource utilization.

In soft real-time systems, however, meeting deadlines is not an absolute requirement (*i.e.*, missing a few deadlines does not severely degrade system performance or cause catastrophic failure). In such systems, a guaranteed minimum probability of completing by the deadline is sufficient. Therefore, there is considerable latitude in such

systems for improving resource utilization and performance as compared with hard real-time systems, through the use of more realistic execution time assumptions.

Given probability distribution functions (PDFs) representing tasks' execution time requirements, and tasks' communication and precedence requirements, represented as a directed acyclic graph (DAG), this dissertation proposes and investigates algorithms for constructing non-preemptive stochastic schedules. New PDF manipulation operators developed in this dissertation are used to compute tasks' start and completion time PDFs during schedule construction. PDFs of the schedules' completion times are also computed and used to systematically trade the probability of meeting end-to-end deadlines for schedule length and jitter in task completion times.

Because of the NP-hard nature of the non-preemptive DAG scheduling problem, the new stochastic scheduling algorithms extend traditional heuristic list scheduling and genetic list scheduling algorithms for DAGs by using PDFs instead of fixed time values for task execution requirements. The stochastic scheduling algorithms also account for delays caused by communication contention, typically ignored in prior DAG scheduling research.

Extensive experimental results are used to demonstrate the efficacy of the new algorithms in constructing stochastic schedules. Results also show that through the use of the techniques developed in this dissertation, the probability of meeting deadlines can be usefully traded for performance and jitter in soft real-time systems.

DEDICATION

I dedicate this dissertation to my mother, Suman Dandass, and to the memory of my father, Flight Lieutenant Tanmaya Singh Dandass.

ACKNOWLEDGEMENTS

I wish to acknowledge here all those persons who have encouraged, supported, mentored, challenged, and helped me during my years as a student at Mississippi State University.

I respectfully offer my deepest gratitude to my major professor and dissertation director Dr. Anthony Skjellum for helping me focus on my research goals when needed yet giving me the freedom to let me develop my own research interests. I am also grateful for the financial support and research opportunities he has provided through my appointment as a research associate in the High Performance Computing Laboratory (HPCL). His willingness to actively involve me in writing proposals, research reports, and articles, and by providing opportunities for presenting technical talks to my peers has taught me many of the skills I will rely on in my career as a researcher.

I also offer respectful thanks to Dr. Susan Bridges who has patiently helped me navigate the rules and regulations of the Department of Computer Science and Engineering and the Office of Graduate Studies at Mississippi State University. I also gratefully acknowledge the seemingly countless times she has interrupted her work in order to answer my questions and to provide guidance and encouragement during my studies. Dr. Donna Reese has also been a valuable resource during my professional development as a university-level teacher. I am grateful for the feedback and suggestions

she has provided over the years in helping me improve my teaching style and in helping me develop skills for effectively teaching dry and difficult material.

I am also grateful to the remaining members of my dissertation committee for their help in guiding my research and for the valuable suggestions that have served to improve this dissertation. In particular, I would like to thank Dr. Eric Hansen for suggesting the ideas that have resulted in the development of the estimate-based list scheduling methods investigated in this dissertation. Dr. Arkady Kanevsky has provided early guidance in the development of my scheduling research ideas and was instrumental in shaping the hypothesis of this dissertation. Dr. Raghu Machiraju's emphasis on practical applications for my research has kept this dissertation from becoming overly abstract.

I am also indebted to the members of the faculty in the Department of Computer Science and Engineering for their encouragement and kind works of motivation that have helped me remain focused on my goal of completing this degree during difficult periods. I also offer a special thanks to Dr. Julia Hodges, who has provided financial support and teaching opportunities by employing me as a part-time lecturer in the department.

I wish to acknowledge all my colleagues and students in the HPCL who have contributed valuable ideas during long discussions and heated arguments about the merits of a variety of technologies and techniques. I would like to especially acknowledge Dr. Purushotham (Puri) Bangalore, Vinod Valsalam, and Manoj Apte in this regard. A special "thank you" also goes out to Vijay Velusamy who helped diagnose and correct

difficulties with the computer equipment while I was conducting scheduling experiments. I acknowledge the friendship of Srigurunath (Ecap) Chakravarthi, Jothi Neelamegam, Shane Hebert, and Gerg Henley.

The willingness of Ms. Brenda Collins and the resourceful office staff in the Department of Computer Science and Engineering to handle all the "last-minute" requests and paperwork deserves a special mention here. Their cooperation and friendly assistance has been helped smooth my time at Mississippi State University.

On a personal note, I wish to express my sincere and deep gratitude to my family for supporting my academic degree goals. I owe a special thanks to my wife, Wendy, without whose reservoir of love, encouragement, spiritual support, and understanding (and occasional prodding), this dissertation would never have been completed. I would also like to thank my mother who had the foresight to inculcate in me the value of advanced education and then had the courage and faith to allow me to leave my home in India in order to independently pursue my educational ambitions in the US.

Finally, I would like to acknowledge the following agencies and organizations that have partially funded my research at Mississippi State over the last few years:

- DoE – LLNL, grant 10605-001-00-35.
- US Navy – Through NASA, Stennis, grant NAS1398033DO111.
- NASA, JPL, grant 1221287 01010075.
- NSF, grant CCR9900524 00030264.
- MPI Software Technology, Inc., grant 01061201AH 01050401.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

LIST OF SYMBOLS, ABBREVIATIONS, AND SPECIAL NOMENCLATURE

Symbols	
$\langle (x_1, r_1), \cdots, (x_n, r_n) \rangle$	A PDF specified by specific mappings of domain values (*i.e.*, time) to range values (*i.e.*, probabilities).
$\prec$	Precedence relation; $J_a \prec J_b$, specifies that task J_a must complete before task J_a is released.
$\prec\!\!\succ$	Exclusion relation; $J_a \prec\!\!\succ J_b$ specifies that instances of tasks J_a and J_b cannot preempt each other.
$\oplus$	The PDF translation operator. $\pi_X(x) \oplus k$ implies that the resulting PDF is computed from the original PDF by adding k units to each domain value in π_X.
$\otimes$	PDF convolution operator; $s_i(t) \otimes w_i(\tau)$ indicates the convolution of PDFs $s_i(t)$ and $w_i(\tau)$.
$\Im^+$	The set of positive integers.
$\Re^+$	The set of non-negative real numbers
ϕ_I	Phase of a periodic or sporadic task J_i.
$\lambda_{pc}(\mathcal{G})$	The length of a planning cycle for a set of periodic tasks $\mathcal{G}$.
(v_a, v_b)	An edge in a DAG designated by the originating vertex v_a and destination vertex v_b.
$\lambda_{\mathcal{G}}$	The LCM of task periods in a set of tasks $\mathcal{G}$.
$[l_{fi}, u_{fi}]$	The interval over which the PDF for the finish time PDF for task J_i is defined.
$[l_{si}, u_{si}]$	The interval over which the PDF for the starting time PDF for task J_i is defined.
$[l_{wi}, u_{wi}]$	The interval over which the PDF for the execution time requirement (weight) PDF for task J_i is defined.
$[l_X, u_X]$	The interval over which the PDF for random variable X has non-zero probability.
$c_i(t)$	The execution time requirement remaining for task J_i at time t.
D_i	Relative deadline of task J_i.
d_i	Deadline of a task.
E	The set of edges in a DAG.
$\mathrm{E}[\pi_X(x)]$ or $\mathrm{E}[X]$	Expected value of random variable X.
E_i	Tardiness of a task; the time by which a task exceeds its deadline.

Symbols (continued)	
e_i	An edge in the set of edges in a DAG.
$\bar{e}_i$	Effective execution time for task J_i in the UDA algorithm.
$F^{-1}_{schedule}(x)$	The inverse of the completion time CDF of a schedule.
f_i	Finish time of a task.
$f_i(t)$	The finish time PDF of task J_i.
$F_i(t)$	The finish time CDF of task J_i.
$f_{schedule}(t)$	The completion time PDF of a schedule.
$F_{schedule}(t)$	The completion time CDF of a schedule.
G	A DAG.
L_i	Lateness of a task; the difference between a task's deadline and finish time.
l_X	The lower bound of the interval over which the PDF for random variable X is defined.
M	Schedule length.
$M(x)$	Stochastic schedule length.
P	The set of processors onto which the DAG is to be scheduled.
$P(A)$	The probability with which event A occurs.
p_i	The i^{th} processor in the set of processors onto which the DAG is to be scheduled.
$QoS(J_i)$	A function used to determine the probability with which task J_i is admitted using SRMS.
r_i	Release time for a task J_i.
R_i	Response time for task J_i.
$r_i(t)$	The release time PDF of task J_i.
s_i	Start time of a task.
$s_i(t)$	The start time PDF of task J_i.
$S_i(t)$	The start time CDF of task J_i.
$S_i(t)$	The release time CDF of task J_i.
t_c	Sum of completion times of all tasks in a schedule.
T_i	Period of a periodic task J_i.
$\bar{t}_r$	Average response time of a schedule.
t_w	Weighted sum of completion times of all tasks in a schedule.
U	Processor utilization for periodic schedules on uniprocessor systems
$\tilde{U}$	Stochastic utilization metric
U^*_A	Breakdown processor utilization of periodic scheduling algorithm A.
$u_i(x)$	Utilization demand function for task J_i in the UDA algorithm.
u_X	The upper bound of the interval over which the PDF for random variable X is defined.

Symbols (continued)	
V	The set of vertices in a DAG.
v_i	A vertex in the set of vertices in a DAG.
w_i	Execution time (or weight) of task J_i.
$w_i(t)$	The execution time requirement (or weight) PDF of task J_i.
$W_i(t)$	The execution time requirement (or weight) CDF of task J_i.
w_{vi}	Execution time requirement of vertex v_i.
X_i	Laxity of a task; the amount of time a task can exceed its planned execution time requirement before missing its deadline.
$\zeta(x)$	Schedule compression metric
H	Throughput of a schedule.
$\xi(x)$	QoS-performance tradeoff metric
$\pi_{\max(X1, X2)}(x)$	The maximum PDF operator.
$\pi_{\min(X1, X2)}(x)$	The minimum PDF operator.
$\pi_X(x)$	The probability distribution for random variable X.
$\Pi_X(x)$	The cumulative distribution function for the random variable X.
$\Psi(sched_1, sched_2)$	Relative schedule length improvement metric
$\mathcal{J}$	Set of n tasks, $\{J_1, J_2, ..., J_n\}$, to be scheduled.

Nomenclature	
Exact Method	A method for computing tasks' start and completion time PDFs in a stochastic schedule that uses tasks' execution time requirement PDFs and PDF operators.
Exact SCP	The version of the SCP algorithm that uses the PDF operators at each scheduling step.
Exact SETF	The version of the SETF algorithm that uses the PDF operators at each scheduling step.
Exact SHLEFT	The version of the SHLEFT algorithm that uses the PDF operators at each scheduling step.
Admission Test	Schedulability analysis performed before allowing at task to execute in order to ensure that the real-time system will meet all required deadlines. Tasks failing the admission test are not admitted into the system.
Aperiodic task	A task that is not invoked repeatedly in a system.
B-level	The B-level (or bottom level) of a vertex in a DAG is the longest path from the vertex to a terminal vertex.
Branch and Bound	A systematic search technique used to solve combinatorial optimization problems. The "branch" step expands the scope of solution space searched while the "bound" step prunes regions of the search space that will not lead to the optimal solution.

Nomenclature (continued)	
Breakdown Processor Utilization	The upper bound on the processor utilization within which a scheduling algorithm can guarantee a feasible schedule for an arbitrary set of periodic tasks.
Collision	The loss of communication that occurs when the signals of simultaneous overlapping transmissions are scrambled.
Computation-to-communication ratio	The ratio of average vertex weight to average edge weight of a DAG.
Confidence Level	The confidence level of an assertion is the probability that the assertion is true all the time.
Congestion	A condition that occurs when the network capacity is insufficient to handle the traffic being inserted into the communication network by all the applications.
Constant Bandwidth Server	A periodic scheduling scheme that isolates tasks with variable execution time from each other by guaranteeing that each task will be granted a pre-assigned fraction of the total processor bandwidth.
Cumulative Distribution Function	A function that maps a positive time value to a positive real number representing the sum of probabilities that an event occurs at or before each time value.
Deadline	The time relative to the beginning of the schedule within which a task or schedule must complete in order to meet real-time constraints.
Deferrable Server	A server that reserves processing capacity, as opposed to a fixed interval of time, in a periodic schedule for executing sporadic and aperiodic tasks.
Deterministic Schedule	A schedule in which the starting and completion time of tasks are fixed.
Directed Acyclic Graph	A representation of a parallel application in the form of a graph consisting of vertices that represent computation tasks and edges that represent communication and precedence relations between the vertices. The direction of the edges represents the direction of data flow or precedence between vertices.
Dominant Sequence Clustering	An LS heuristic that uses the b-level and t-level attributes of the vertices in a DAG to determine the critical path of the partially scheduled DAG and gives priority to vertices on the critical path.
Dynamic Critical Path	A LS heuristic that recomputes the critical path of the partially scheduled DAG at each step and schedules vertices on the critical path first. Vertices are scheduled on the processor that minimizes communication cost with predecessor and successor vertices.

Nomenclature (continued)	
Dynamic Scheduling	A scheduling paradigm in which scheduling decisions are made based on the varying requirements of a changing workload.
Earliest Critical Deadline First	A modification of the EDF algorithm in which the deadline of a task instance that has not completed within its deadline is modified to become the deadline of the next instance of the task.
Earliest Time First	A LS heuristic that prioritizes vertices in non-decreasing order of their earliest possible starting times.
Edge Zeroing	A LS heuristic that strives to reduce communication costs by allocating vertices connected by large edges onto the same processor.
End Time	Synonymous with "finish time."
End-to-end deadline	The deadline by which all tasks in the DAG (or the corresponding schedule) must complete.
Estimate Method	A method for computing tasks start and completion time PDFs by constructing an initial deterministic schedule using estimated fixed values to represent each task's execution time requirements, and the using the deterministic schedule to construct the final stochastic schedule.
Estimate SCP	The version of the SCP algorithm that uses fixed task weight estimates to construct an initial schedule before using PDF operators to construct the final schedule.
Estimate SETF	The version of the SETF algorithm that uses fixed task weight estimates to construct an initial schedule before using PDF operators to construct the final schedule.
Estimate SHLEFT	The version of the SHLEFT algorithm that uses fixed task weight estimates to construct an initial schedule before using PDF operators to construct the final schedule.
Expected Value	The mean value of a random variable taken over an infinitely large sample.
Finish Time	The time relative to the beginning of the schedule at which a task completes execution.
Genetic Algorithm	An optimization algorithm based on the principle of natural selection.
Genetic List Scheduling	A hybrid GA and LS approach to scheduling DAGs in which the GA determines the priority in which tasks are scheduled using the LS approach.
Hard Real-time	A real-time system or task that must complete within its deadline with 100% probability in order to avoid catastrophic failure.

Nomenclature (continued)	
Heuristic Scheduling	A scheduling paradigm in which the system strives to achieve optimality but does not guarantee it.
Highest Level First with Estimated Times	A LS heuristic that prioritizes vertices according to non-decreasing order of their b-level attributes.
Immediate Predecessor Tasks	The immediate predecessor tasks of a task J_i in a DAG are those tasks that are directly connected to J_i and are followed by J_i.
Immediate Successor Tasks	The immediate successor tasks of a task J_i in a DAG are those tasks that are directly connected to J_i and follow J_i
Independent Random Variables	Random variables are mutually independent if the observation of any particular value of one variable has no influence on the probability of observing any value of the other variables.
Jitter	The variance in the execution time requirement of tasks in a real-time system
Lateness	Difference between a task's deadline and finish time.
Linear Clustering	A LS heuristic that assigns vertices in the critical path of a DAG to the same processor in order to reduce communication costs.
List Scheduling	A class of heuristic DAG scheduling algorithms in which ready tasks are scheduled in the order determined by one of a variety of heuristics.
Mobility Directed	A LS heuristic that prioritizes vertices in non-decreasing order of their relative mobility attribute.
Modified Critical Path	A LS heuristic that priorities vertices in non-decreasing order of the latest time when they can be started without extending the schedule length
Non-preemptive scheduling	A scheduling paradigm in which the currently executing task cannot be interrupted in order to allow another task to execute.
Offline Scheduling	A scheduling paradigm in which scheduling decisions are made before the system is executed.
Online Scheduling	A scheduling paradigm in which scheduling decisions are made while the system is executing.
Optimal Scheduling	A scheduling paradigm in which a cost function is minimized or a benefit function is maximized
Parallel and Distributed Real-time System	Parallel and distributed real-time systems exploit the inherent concurrency in applications in order to reduce execution time by apportioning the workload between several processors while striving to retain the application's predictability requirements.
Period	The length of intervals between successive activations of a periodic task.
Periodic schedule	A schedule for a periodic system
Periodic system	A system consisting of periodic tasks
Periodic task	A task that is repeatedly executed at a fixed rate

Nomenclature (continued)	
Phase	The time relative to the beginning of the schedule when the first instance of a periodic or sporadic task is released.
Planning Cycle	The minimum length schedule required in order to schedule all tasks in a periodic system. The schedule in the planning cycle is repeatedly executed back-to-back over the lifetime of a periodic system.
Polling Server	A special periodic task that is used to reserve intervals of time in a periodic real-time schedule for executing sporadic and aperiodic tasks.
Predictability	Predictability is a property of real-time systems that implies that the runtime behavior of the system is repeatable.
Preemption	Interrupting the currently executing task in order to execute another task. The interrupted task may be allowed to resume at a later time.
Preemptive scheduling	A scheduling paradigm that permits an executing task to be interrupted in order to allow another (typically higher priority) task to execute
Priority	A quantitative attribute of a task describing its importance relative to other tasks.
Probabilistic Time Demand Analysis	A preemptive stochastic scheduling algorithm for periodic stochastic tasks that does not account for precedence relations between tasks.
Probability Distribution Function	A function that maps a positive time value to a positive real number. The real number indicates the probability with which an event occurs at each time value.
Processor Utilization	The fraction of time in a uniprocessor schedule when a processor is not idle.
QoS-Performance Tradeoff Metric	This metric relates the reduction in the required probability of meeting end-to-end deadlines to the resulting schedule compression
Quality of Service	A generic term used to describe the level of assurance a system provides users about the predictability of offered services
Rate Monotonic Scheduling	A dynamic online scheduling algorithm for periodic tasks that assigns higher priority to tasks with shorter periods.
Ready time	Synonymous with “release time.”
Real-time System	A systems that is required to respond to external stimulus within a guaranteed period of time.
Relative deadline	The amount of time relative to the release time of a task within which the task must complete.

Nomenclature (continued)	
Relative Schedule Length Improvement Metric	This metric is the relative reduction in the stochastic schedule length of one schedule relative to the stochastic schedule length of another schedule.
Release time	The time relative to the beginning of the schedule when a task becomes ready to execute. Synonymous with "read time."
Response time	The difference between the finish time and release time of a task.
Schedule Compression Metric	The relative reduction in the width of the schedule completion PDF when the required probability of meeting end-to-end deadlines is reduced below 100%.
Schedule Length	The number of time units relative to the start of the schedule required for completing all tasks in the schedule.
Simulated Annealing	A heuristic combinatorial optimization technique based on the physical process of heating and then slowly cooling a substance to obtain strong crystallization structures.
Slot-fitting Threshold	The minimum probability with which a task must fit in a slot in a schedule in order to permit the insertion of the task into the slot.
Soft real-time	A real-time system or task that can miss deadlines occasionally without resulting in catastrophic failure.
Sporadic Task	A sporadic task repeats that repeats at irregular intervals. The length of the intervals is bounded from below, thereby restricting the frequency at which the aperiodic task repeats.
Start Time	The time relative to the beginning of the schedule at which a task begins execution.
Static Scheduling	A scheduling paradigm in which the scheduling decisions are made based on a fixed workload.
Statistical Rate Monotonic Scheduling	An extension to RMS that accounts for variable execution time requirements of tasks.
Stochastic Critical Path	The new LS-based stochastic scheduling algorithm developed in this dissertation that uses PDF operators in order to determine the stochastic critical path of the schedule and give priority to vertices on the critical path. The algorithm also uses the PDF operators to allocate resources to tasks.
Stochastic Earliest Time First	The new LS-based stochastic scheduling algorithm developed in this dissertation that uses PDF operators in order to prioritize ready vertices according to their earliest expected execution time and uses the PDF operators to allocate resources to tasks.
Stochastic Footprint Metric	The sum of the count unit time slots per resource in the schedule during which the resource is reserved for execution the any of the schedule's tasks with non-zero probability.

Nomenclature (continued)	
Stochastic Highest Level First with Estimated Time	The new LS-based stochastic scheduling algorithm developed in this dissertation that uses the tasks expected b-level values to prioritize vertices, but uses PDF operators instead of fixed value operators to allocate resources to tasks.
Stochastic Jitter	The variance in the completion time of a task in a stochastic schedule cause by the variance in the task's execution time requirements and the variance in the task's starting time caused by the variance in the completion time of preceding tasks.
Stochastic Schedule	A schedule in which the starting time and completion time of tasks is specified in the form of probability distribution functions, as opposed to fixed values.
Stochastic Schedule Length	The minimum amount of time required for completing a stochastic schedule with a given probability.
Stochastic Time Demand Analysis	An extension to PTDA that accounts for tasks with deadlines greater than their periods and also accounts delays caused by contention over shared resources.
Stochastic Utilization Metric	A measure of the resource utilization of the stochastic schedule.
Terminal Tasks	Tasks in a schedule that are not followed by other tasks.
T-Level	The T-level (or top level) of a vertex in a DAG is the longest path from an entry vertex to this vertex.
Total Bandwidth Server	A server that dynamically assigns deadlines to periodic tasks in a periodic real-time system.
Translation Lookaside Buffer	A special cache memory that holds frequently used page table entries and is used for speeding the translation of logical addresses to physical addresses in advanced architecture processors that support paged memory.
Utilization Demand Analysis	An admission control technique for preemptive periodic scheduling for tasks with variable execution time requirements based on computing the overall utilization demands of a set of tasks.

Abbreviations	
BB	Branch and bound
CBS	Constant bandwidth server
CCR	Computation-to-communication ratio
DAG	Directed acyclic graph
DCP	Dynamic critical path
DMA	Direct memory access

Abbreviations (continued)	
DMS	Deadline monotonic scheduling
DS	Deferrable server
DSC	Dominant sequence clustering
ECDF	Earliest critical deadline first
EDD	Earliest due date
EDF	Earliest deadline first
ETF	Earliest time first
EZ	Edge zeroing
FFT	Fast Fourier transform
GA	Genetic algorithm
GLS	Genetic list scheduling
HFJ	Hierarchical fork join
HLEFT	Highest level first with estimated times
I/O	Input/output
LC	Linear clustering
LCM	Least common multiple
LS	List scheduling
MCP	Modified critical path
MD	Mobility directed
MVA	Mean value analysis
NIC	Network interface card
OUT	Out tree
PDF	Probability distribution function
PS	Polling server
PTDA	Probabilistic time demand analysis
QoS	Quality of service
RMS	Rate monotonic scheduling
SA	Simulated annealing
SCP	Stochastic critical path
SETF	Stochastic earliest time first
SFJ	Simple fork join
SHLEFT	Stochastic highest level first with estimated times
SRMS	Statistical rate monotonic scheduling
STDA	Stochastic time demand analysis
TBS	Total bandwidth server
TLB	Translation lookaside buffer
UDA	Utilization demand analysis
WCET	Worst case execution time

CHAPTER I

INTRODUCTION

This chapter introduces basic real-time concepts and describes the fundamental problem of constructing non-preemptive schedules for soft real-time parallel applications with non-deterministic computation requirements times and arbitrary precedence constraints. The specific problem of emphasis in this dissertation is motivated here and the main hypothesis of the research is presented in this chapter. The approach used to solve the problem, the plan for experiments to validate the scheduling techniques, and the expected contributions of the research are also summarized here.

1.1 Real-Time Systems

A real-time computer system is one that guarantees that its component tasks will begin and complete execution within a predefined interval of time [26, 31, 147]. Therefore, the correctness of a real-time system depends both on the accuracy of computations and the time at which the system begins and completes those computations. Real-time systems typically must react to external events within specific time constraints. However, real-time systems are not necessarily equivalent to *fast* systems; fast is a relative term that does not completely express the timing characteristics required of real-time systems.

The primary distinguishing feature between high-performance (*i.e.*, fast) computing systems and real-time systems is that the former emphasizes throughput and reducing average response times and the latter emphasizes timeliness and *predictability* of completion times [147]. Predictability implies that the timing characteristics of tasks are deterministic and repeatable so as to enable scheduling that meets timing constraints. High performance systems reduce average response times by utilizing techniques such as time slicing, memory hierarchies (*e.g.*, caches), and speculative execution. However, these techniques reduce the predictability of task runtimes, making it difficult to construct schedules that guarantee that individual tasks will meet their timing constraints. Therefore, the focus of real-time system design is on improving *predictability*.

However, predictability is not the only critical factor in determining the success of a real-time system design and implementation. For example, a real-time control system must be designed to react within the timing characteristics of the system being controlled [147]. In particular, a real-time flight control system generally requires sub-second response time to pilot input, whereas a meteorological forecasting system has several minutes or hours available to it to respond to changes in atmospheric conditions. This example illustrates that timing constraints in practical real-time systems cannot be arbitrarily extended to ensure task completion within deadlines. Also, utilizing faster processors also does not automatically guarantee that tasks will meet timing requirements because interaction of tasks with each other and with the environment have to be taken into consideration under realistic loads.

Improved hardware performance is also enabling system designers to timeshare the hardware between several competing real-time tasks, thereby reducing the amount of processing time that can be dedicated exclusively to a single task. Effective timesharing is especially critical in applications where restrictive weight and power budgets prohibit the dedicated allocation of hardware resources to tasks [108].

Computation requirements in many real-time application domains (*e.g.*, signal processing) exceed the computation capacity of a single processor, and therefore, require *parallel and/or distributed real-time* processing. Parallel and distributed real-time computing exploits the inherent concurrency in applications in order to apportion the workload between several processors while striving to retain such applications' predictability requirements.

In order to achieve a balance of timely and predictable performance, real-time systems typically use specialized schedulers in order to control when tasks are executed with the goal of meeting timing requirements. Essentially, the real-time system strives to satisfy the quality-of-service (QoS) demands imposed by real-time tasks. In the past, scheduling policies have been based on labor-intensive, *ad hoc*, low-level optimization techniques [26, 31, 147]. These techniques have included:

- utilizing hand-optimized assembly language routines,
- utilizing priority-based interrupt handling,
- introducing simplifying (and not necessarily correct) timing assumptions, and
- performing extensive simulations to verify that timing constraints are met for a set of expected scenarios.

The use of *ad hoc* techniques to construct a real-time system requires exhaustive testing in order to validate the system's correctness every time its hardware, software, and/or constraints are modified. If this testing and verification do not cover all possible combinations and sequences of external events and internal scheduling decisions, nor account for all uncertainties, the system can fail under conditions that have not been previously encountered.

1.2 Classification of Real-time Tasks and Systems

The consequences of failing to comply with timing constraints are used to broadly classify real-time tasks and systems [26]. The failure to meet a *hard* real-time constraint results in catastrophic consequences (*e.g.*, loss of life and/or property) and invalidates the correctness of a real-time system. Tasks and systems with hard real-time constraints are said to be hard real-time tasks and hard real-time systems, respectively. Examples of hard real-time systems include:

- chemical and nuclear plant control,
- automotive applications,
- medical applications, and
- flight control systems.

Techniques for constructing hard real-time systems typically utilize estimates of the real-time tasks' worst-case execution time (WCET) in scheduling decisions in order to ensure timely execution. However, estimating WCET accurately is difficult because of the complex interaction of software and hardware subsystems. For example, the time

taken to complete a task can vary from run to run on modern advanced architecture processors incorporating instruction and data caches, and pipelined and out-of-order speculative execution with branch prediction. This variance, also called *jitter* [26], is further exacerbated by the interference of interrupt handling and direct memory access (DMA) operations.

The typical approach to determining WCET involves analyzing the compute time of sequences of instructions under simplified worst-case assumptions (*e.g.*, instructions and data are not resident in cache, branch prediction tables are invalidated, all expected DMA operations and interrupts will occur, etc.) [71, 103, 105, 106, 110]. Analyzing only the worst-case scenario, instead of the expected sequences of task executions, simplifies schedule construction because the cost of analyzing all expected combinations of software and hardware interactions can be prohibitive. This simplification, used to facilitate schedule construction, however, results in significantly overestimated task execution times relative to the actual observed execution time distribution. In systems where worst-case situations occur infrequently, the use of WCET in scheduling decisions results in resource under-utilization because reserved resources are left idle for a significant fraction of the time in order to guarantee their availability when needed.

Furthermore, in order to reduce a hard real-time task's execution time jitter caused by interrupts and direct memory access (DMA) resulting from asynchronous input/output (I/O) operations, hard real-time system schedulers typically isolate and serialize computation and I/O bound tasks [30]. However, this restriction on when DMA and interrupts are permitted to occur also results in resource under-utilization because I/O

devices are idled when a compute-intensive task is executing and the processor is idled when an I/O operation is underway.

By contrast, a *soft* real-time constraint can be violated without resulting in catastrophic consequences and need not jeopardize the correctness of the real-time system [26, 31]. The value of results from a soft real-time system depends on the system's ability to meet constraints, and therefore, compliance with deadlines with a specified minimum probability is required for such systems. Tasks and systems with soft real-time constraints are said to be soft real-time tasks and soft real-time systems, respectively. Examples of soft real-time systems include:

- mobile (cellular) telecommunications,
- multimedia applications, and
- interactive systems such as flight simulators and video games.

Because soft real-time systems can tolerate occasional late tasks, these systems are typically designed to provide statistical timing guarantees. The use of statistical guarantees as opposed to absolute guarantees enables the use of more optimistic timing assumptions instead of WCET in the schedule construction process, resulting in more efficient systems. In many soft real-time systems (*e.g.*, variable bit rate multimedia applications [9]), the execution time of tasks can be specified in terms of discrete probability distributions of execution times determined either analytically or empirically [33, 34, 61, 83, 152]. The probability distribution of a task is derived from the histogram of expected frequencies of execution times of individual instances of the task.

1.3 Real-time Scheduling

Scheduling is the process of allocating limited system resources to tasks in order to meet the timing constraints of the system. Scheduling research in Computer Science and in Operations Research has traditionally focused on improving average values of performance metrics such as response time, throughput, average completion times, and cost [96]. Traditional scheduling techniques applied in Operations Research frequently rely on results from asymptotic analysis of simplified and relatively uniform statistical task execution models that do not reflect realistic real-time task execution requirements (*e.g.*, in [24]), and therefore, cannot be directly applied to guarantee real-time constraints. Furthermore, combinatorial optimization scheduling techniques developed in Operations Research typically do not consider recurring, synchronizing, and communicating tasks, making these techniques impractical for real-time scheduling.

A number of deterministic uniprocessor scheduling techniques specifically designed for *periodic* real-time systems have been proposed over the last few decades [73, 78, 97, 99, 101, 109, 145]. A periodic system is one in which each task has a periodic recurrence (*i.e.*, instances of each task are repeatedly executed at a fixed rate). Note that different tasks in a periodic system may have different periods. A variety of scheduling techniques utilizing statistical behavioral characteristics of tasks have also been proposed for soft real-time systems [10, 61, 83, 152]. These approaches provide analytical techniques for computing the probability that tasks in a soft real-time application will meet deadlines. The following is a classification of real-time scheduling

algorithms (and resulting schedules) based on the properties of the task set under consideration and the objectives of the scheduling algorithm [26]:

- *Preemptive vs. non-preemptive*. A preemptive scheduling algorithm can interrupt a running task to execute another ready task on the same processor. The interrupted task is resumed when the interrupting task completes. Conversely, a non-preemptive scheduling algorithm must wait for the currently executing task to complete before executing another ready task on the same processor.
- *Static vs. Dynamic*. A static scheduling algorithm is one that makes scheduling decisions based on a fixed workload wherein all the tasks and task properties are known before the tasks are executed. In dynamic scheduling, the task set to be scheduled at any given time is unknown a priori and can change over the system's lifetime.
- *Offline vs. Online.* In an offline scheduling algorithm, scheduling decisions for the entire task set are made before the system is started. The resulting schedule essentially consists of a calendar (*i.e.*, sequence of task activation times) that ensures that all real-time constraints are met. An online scheduling algorithm actively makes scheduling decisions when a currently executing task completes or a new task becomes ready for execution.
- *Optimal vs. Heuristic.* An optimizing scheduling algorithm minimizes a cost function (*e.g.*, the number of tasks violating constraints) or maximizes a benefit function defined over the system's tasks. A heuristic algorithm, on the other hand, strives to achieve, but does not guarantee, optimality.

Real-time systems in which the exact workload characteristics are not known at design time typically utilize dynamic online scheduling (*e.g.*, executing tasks with the earliest deadlines first [73]). Such algorithms are also typically preemptive in order to accommodate newly released tasks whose deadlines are earlier than currently executing tasks. In those real-time systems where the workload is well defined, a schedule can be constructed offline and stored in a table before the system begins execution. Such table-driven scheduling increases the efficiency of the real-time system because the runtime task dispatcher only needs to look up previously made scheduling decisions instead of performing scheduling related computations every time a scheduling decision needs to be made. However, table-driven, offline schedules are static in nature because they assume task sets have well-defined runtime characteristics and cannot adapt to changing runtime requirements of the applications.

When tasks in a real-time application cannot be preempted and have non-synchronous release times (*i.e.*, all tasks are not ready to execute at the same time), the problem of constructing optimal schedules becomes NP-hard, in general [100]. In the context of real-time systems, the primary scheduling goal is to minimize the number of tasks that miss deadlines; reducing schedule lengths is of secondary concern. Given the intractable nature of this problem, simple heuristic techniques are often used to construct schedules in a reasonable amount of time in online systems where scheduling decisions must be made quickly (*e.g.*, scheduling tasks with the earliest deadline first). However, for many such problems, simple heuristics are known to produce sub-optimal solutions [26]. The use of more complex techniques and heuristics in order to produce optimal or

near-optimal schedules increases the computation time required and this limits their usefulness in online scheduling [22, 91, 158]. Therefore, offline techniques are typically used to create non-preemptive schedules.

Preemptive uniprocessor real-time scheduling algorithms have also been extended to perform scheduling for hard and soft real-time distributed and parallel processing systems. For example, in order to provide predictable delivery of packets over packet-switched networks, traffic volume is shaped using admission control and rate-controlling techniques [163, 164]. Furthermore, network resources are typically either reserved in advance, or allocated to communicating tasks using the traditional preemptive, priority-based scheduling techniques originally developed for periodic processor scheduling [9, 58, 59, 165, 167].

1.4 Motivation

In soft real-time systems, system designers are afforded considerable flexibility in the application of scheduling policies used to balance the need for predictability with the need for improved performance. These systems can improve resource utilization and performance by making scheduling decisions based on the premise that, in a given interval of time, it is unlikely that all successively activated tasks will require their full WCET to complete. This is particularly useful in many real-time control applications in defense and space exploration systems where restrictions on the volume, mass, and energy available for performing computations require the efficient scheduling of several tasks executing concurrently in a time-shared cluster environment, rather than dedicating processors to individual tasks.

Bernat, Burns, and Llamosí [17] provide three example real-time systems where occasional deadline misses can be tolerated.

1. Computer-driven automatic control systems typically oversample the environment by a factor of at least 5 (and up to 40). This overampling implies that deadlines can be missed as long as a large number of back-to-back deadlines are not missed.
2. In many automated monitoring systems, the monitoring period is overestimated or decided by a "rule of thumb." Therefore, an occasional deadline miss can be tolerated as long as the delay in any action that is undertaken in response to the monitoring is bounded.
3. In multimedia systems, video frames are decoded and displayed at a fixed rate. If the system misses a frame-decoding deadline, then either a partial frame is displayed or the frame is skipped entirely. As long as too many frames are not lost, viewers will tolerate the slight degradation in video quality resulting from an occasional deadline miss.

Because addressing the tradeoff between providing QoS and enhanced performance at lower cost through better resource utilization has useful practical applications, the scheduling techniques developed in this dissertation strive to improve resource utilization and performance while bounding the risk of missing deadlines to a tolerable level.

The elimination of overhead caused by unnecessary context switching (*i.e.*, the context switches that result from the scheduling policy used but do not improve schedule quality) is another performance improvement goal of parallel scheduling algorithms [6, 7,

37, 57, 69, 126, 141, 142]. Preemption used in most periodic real-time scheduling algorithms can reduce performance because of additional context switching times, and reduced locality (*e.g.*, cache content and branch prediction table entries setup for the original task are disturbed by the interrupting task). Consequently, the scheduling techniques developed in this dissertation eliminate preemption in direct contrast to most existing real-time scheduling algorithms that rely on preemption to perform scheduling.

Another motivating factor for this research is to investigate and develop scheduling techniques for parallel real-time applications with complex inter-task interactions that are difficult to represent as periodic tasks and are more naturally represented in the form of *directed acyclic graphs* (DAGs) [91]. Data flow patterns for such parallel applications appear as sequences of tasks that branch and merge in arbitrary fashion. Tasks in these applications can be represented in the form of DAGs by restricting loops to exist in single vertices, or by unrolling loops into sequences of several vertices.

Most existing periodic real-time scheduling techniques assume that tasks scheduled to execute on a processor are self-contained and do not interact with other tasks (*i.e.*, there is no provision for observing task ordering or precedence restrictions). Furthermore, the tasks are scheduled using preemption and task priorities are determined by task execution rates or proximity of deadlines. Schedulers in most distributed real-time systems applications also typically require the applications to be organized in the form of chains of tasks, with each task in a chain executing on a separate processor and each processor preemptively executing several tasks from different chains in a time-

shared manner (*e.g.*, [61] and [83]). All tasks in such chains have a single successor task downstream.

The results of this dissertation enable the non-preemptive scheduling of a broad class of parallel soft real-time applications with precedence constraints in a manner that allows a useful and predictable tradeoff between QoS requirements and performance of the applications. The two QoS characteristics of stochastic schedules studied in this dissertation are the probability of meeting the end-to-end deadline and the average jitter factor in the completion time of the tasks in the schedule. The probability of meeting the end-to-end deadline of a schedule is the probability that the tasks in the schedule will complete within the allocated time. The jitter factor of a task is the ratio of completion time jitter and the execution time requirement jitter.

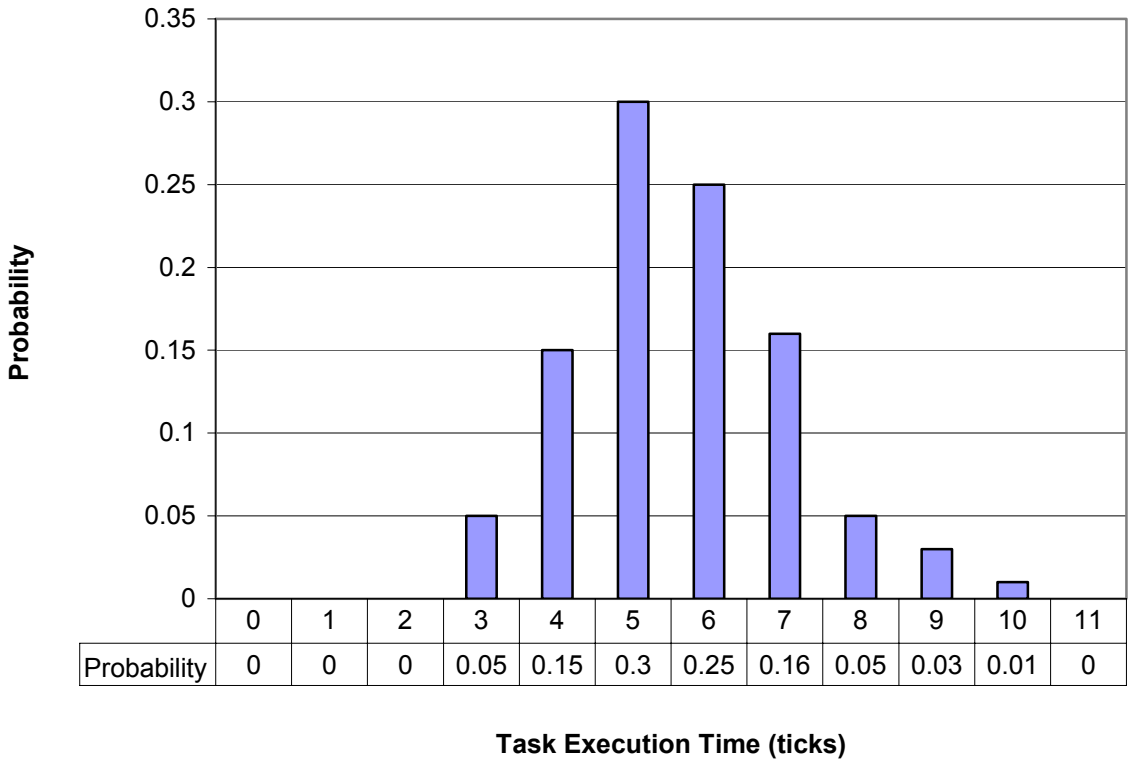
1.5 Hypothesis

The hypothesis of this dissertation is that the probabilities of soft real-time interacting tasks with inter-task precedence constraints completing at specific points in time, when scheduled in a non-preemptive parallel environment, can be computed. These probabilities, in turn, can be effectively applied in combinatorial optimization processes to construct stochastic schedules that map tasks onto identical processors connected by a finite-performance interconnection network such that the probability of meeting end-to-end deadline of the application can be traded off with the following cost metrics:

- *application runtime,*
- *resource utilization, and*

- *completion time jitter.*

In a simple example, Figures 1.1-1.4 together with Table 1.1 illustrate how relaxing stringent timing requirements allows the system to utilize resources more efficiently than possible under hard real-time constraints. In this example, four tasks with variable computation requirements are to be non-preemptively scheduled on a single processor. The computation requirement of a task is given by a *probability distribution function* (PDF). A PDF maps a time quantity representing the execution time requirement of the task to the probability that the task will require that much time to complete.



	0	1	2	3	4	5	6	7	8	9	10	11
Probability	0	0	0	0.05	0.15	0.3	0.25	0.16	0.05	0.03	0.01	0

Figure 1.1 A Representative PDF for Task Execution Time Requirements

This example assumes identical execution time PDFs for all four tasks. The PDF is specified in Figure 1.1. The minimum, nominal, and maximum time requirements are 3, 6, and 10 units, respectively.

Figure 1.2 shows the Gantt chart for the planned hard real-time schedule using WCET estimates, and the actual runtime behavior of the tasks when the tasks only require their nominal and minimum execution times. The WCET schedule requires 40 time units to complete the four tasks and because the maximum required execution time is dedicated to each task, this schedule has a 100% guarantee of completing within the 40 time units. Note that in the WCET schedule, each task is released only after the previous task's maximum guaranteed time slot.

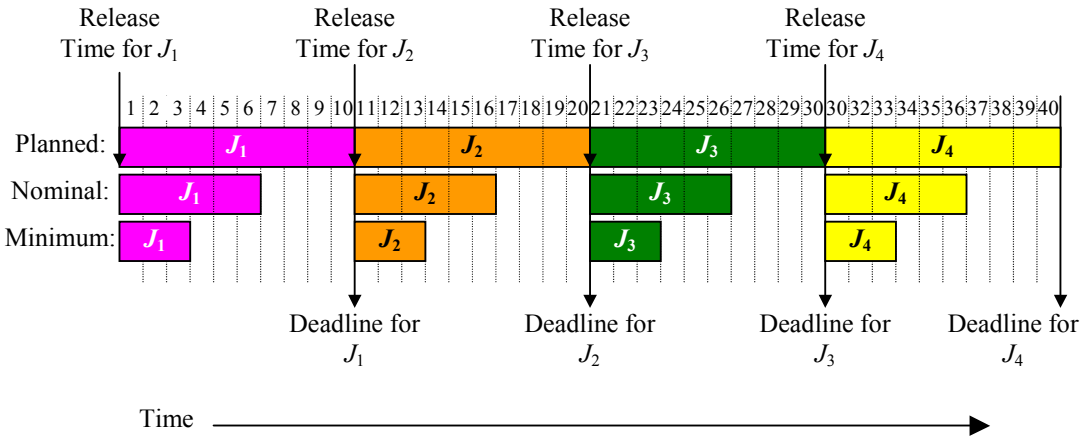


Figure 1.2 A Hard Real-Time Schedule Example

However, when tasks only consume their nominal or minimum execution time, the processor is idle 40% or 70% of the time, respectively. Furthermore, the probability that all four tasks will require the maximum execution time is $(0.01)^4$ or 0.00000001 (*i.e.*,

the product of the individual tasks' probabilities of requiring their maximum execution times). This suggests that, if instead of reserving the maximum execution time for each process, processes are allowed to begin immediately after the preceding task has completed, the resulting schedule may utilize resources better than the WCET schedule.

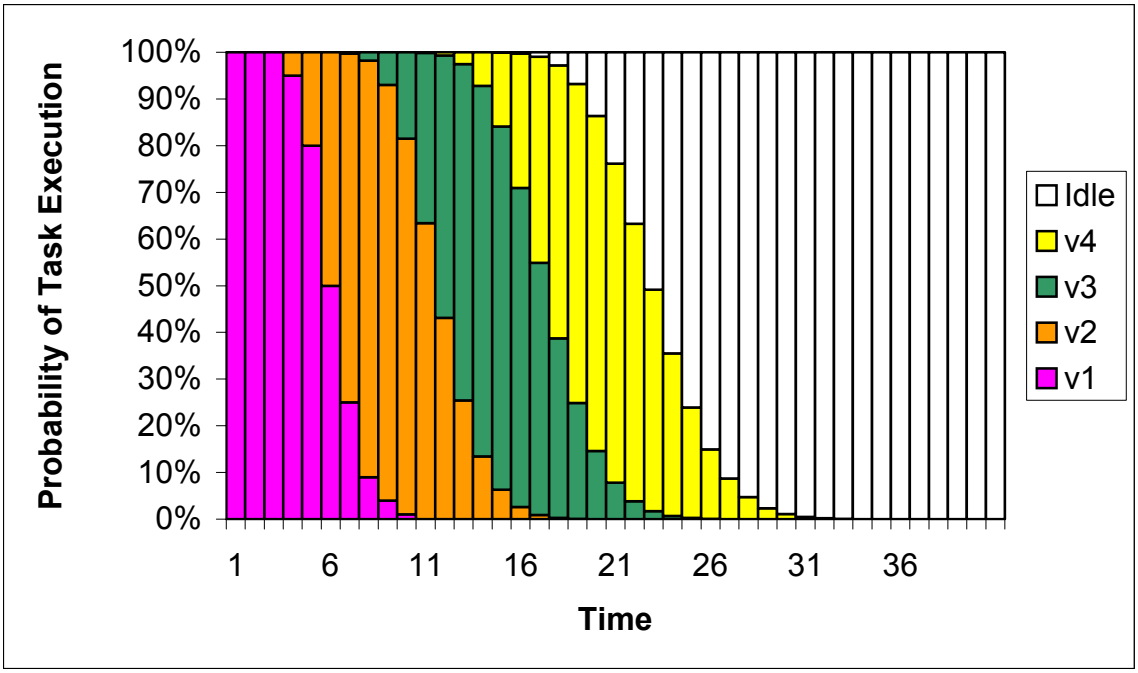


Figure 1.3 Execution Profile for a "Tight" Soft Real-Time Schedule

Figure 1.3 depicts the execution profile for a schedule when the tasks are allowed to begin as soon as the previous task has completed and Table 1.1 lists the start time and completion time PDFs of the tasks in the schedule. Note that the tasks occupy overlapping intervals of potential execution time (*i.e.*, tasks J_1, J_2, J_3, and J_4 may execute during the time intervals [1, 10], [4, 20], [7, 30], and [10, 40], respectively). The intervals represent the range of time units when the tasks may execute.

Table 1.1 Example Start and Completion Time PDFs

	J_1 PDFs		J_2 PDFs		J_3 PDFs		J_4 PDFs		Completion
Time	**Start**	**End**	**Start**	**End**	**Start**	**End**	**Start**	**End**	**Probability**
1	1.0								
2									
3		0.05							
4		0.15	0.05						
5		0.3	0.15						
6		0.25	0.3	0.0025					
7		0.16	0.25	0.015	0.0025				
8		0.05	0.16	0.0525	0.015				
9		0.03	0.05	0.115	0.0525	0.000125			
10		0.01	0.03	0.181	0.115	0.001125	0.000125		
11			0.01	0.203	0.181	0.005625	0.001125		
12				0.1765	0.203	0.01875	0.005625	6.25E-06	6.25E-06
13				0.12	0.1765	0.0462	0.01875	7.50E-05	8.13E-05
14				0.0716	0.12	0.08745	0.0462	0.000488	0.000569
15				0.037	0.0716	0.13155	0.08745	0.00215	0.002719
16				0.0171	0.037	0.160125	0.13155	0.007111	0.00983
17				0.0062	0.0171	0.161715	0.160125	0.01852	0.02835
18				0.0019	0.0062	0.13822	0.161715	0.039203	0.067553
19				0.0006	0.0019	0.102855	0.13822	0.06884	0.136393
20				0.0001	0.0006	0.0678	0.102855	0.101942	0.238334
21					0.0001	0.040186	0.0678	0.129014	0.367348
22						0.021315	0.040186	0.141415	0.508763
23						0.010149	0.021315	0.135996	0.644759
24						0.004343	0.010149	0.116292	0.761052
25						0.001677	0.004343	0.089468	0.85052
26						0.000573	0.001677	0.062531	0.913051
27						0.000165	0.000573	0.039935	0.952986
28						4.20E-05	0.000165	0.023396	0.976382
29						9.00E-06	4.20E-05	0.012603	0.988985
30						1.00E-06	9.00E-06	0.006256	0.995241
31							1.00E-06	0.002862	0.998102
32								0.001203	0.999305
33								0.000463	0.999768
34								0.000162	0.99993
35								5.15E-05	0.999981
36								1.45E-05	0.999996
37								3.52E-06	0.999999
38								7.40E-07	0.999999
39								1.20E-07	0.999999
40								1.00E-08	1.0

Note that only a single task can execute at any given time and that once a task begins execution, it must complete before the next task can begin. Also note that at the extremes of each interval, the probability that the current task will execute is sufficiently

small so as to be indiscernible in Figure 1.3. For example, the probability that task J_3 executes at time 30 is $1E^{-006}$. Although these probabilities are small, they must be accounted for in the schedule in order to ensure accuracy.

Given the PDF of the execution start time of task J_i, the completion time PDF of J_i is computed by the convolution of the starting time PDF and the execution time PDF of J_i [84]. Task J_{i+1} is started immediately after J_i completes and its start time PDF is essentially the completion time PDF of J_i that has been *translated* (*i.e.*, shifted) to the right by one time unit. Note that the start time PDF of J_1 is given by the initial condition that J_1 starts at time 1 with 100% probability. From probability theory, the probability of the schedule completing within an arbitrary deadline of t is given by the sum of probabilities that J_4 completes before or at time t. The sum of completion time probabilities for J_4 is 100% at 40 time units, which corresponds to the length of the WCET schedule. Similarly, the sum of probabilities at 30 time units is 99.52%. Therefore, if a probability of completing within a deadline of 99.52% is required, then the end-to-end deadline for the schedule can be set to 30 time units, reducing the schedule length by 10 time units compared to the WCET schedule. A similar approach for computing task completion time PDFs is proposed in preemptive uniprocessor soft real-time scheduling research for periodic tasks, without considering precedence constraints [10, 61, 152]. The research in this dissertation extends existing approaches by providing techniques and mechanisms for constructing non-preemptive stochastic schedules for parallel soft real-time tasks with precedence constraints. Details of the stochastic scheduling approaches developed in this dissertation are given in Chapter III.

The drawback of the approach used to construct the schedule in Figure 1.3 is that the ranges of completion times of the tasks towards the end of the schedule are substantially wider than the range of their corresponding execution time requirements because of the uncertainty in the completion time of the previous tasks. In order to prevent the previously executing task from interfering with following tasks, the planned start time of each of the following tasks can be intentionally delayed by a small amount. As long as the remaining execution time, after the delay, of each following task is at least as much as the maximum execution time required by the task, the task's completion time jitter is mitigated without increasing the schedule length.

Figure 1.4 shows the execution profile for schedule that results when the release time of each task has been delayed by 50% of the difference between the task's maximum and minimum start time in the schedule from Figure 1.3. Note that a delay of 100% of the difference between minimum and maximum start time will result in the WCET schedule. In Figure 1.4, the total execution time for the schedule remains 40 units. However, the completion time intervals, and the concomitant task completion jitter are also reduced. Specifically, J_1 completes in the interval [3, 10] (same as the original interval); J_2 completes in the interval [9, 20] as opposed to the original interval [6, 20]; J_3 completes in the interval [17, 30] as opposed to the original interval [9, 30]; and J_4 completes in the interval [26, 40] as opposed to the original interval [12, 40].

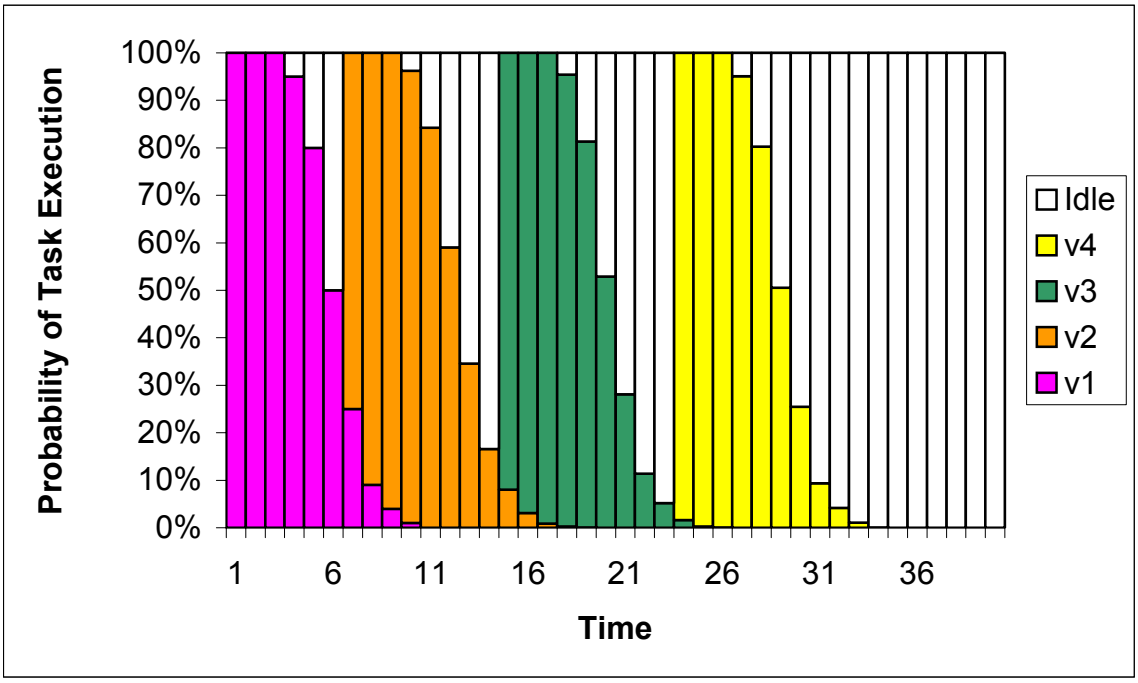


Figure 1.4 Execution Profile for a Soft Real-Time Schedule with Jitter Control

The reduction in jitter comes at a cost of translating the probabilities towards the latter portion of the schedule, thereby decreasing the potential for trading probability of timely completion for schedule length. Specifically, at time 30, the probability of completion for the schedule in Figure 1.4 is 90.64%. At time 33, the probability of completion is 99.97%, the closest probability value greater than or equal to the original target of 99.5%. Therefore, the schedule in Figure 1.4 is slightly longer, by 2 time units, than the schedule in Figure 1.3 if a target probability of completing all four tasks is 99.5%.

1.6 Scheduling Approach and Assumptions

An objective of this research is to develop non-preemptive stochastic scheduling techniques for soft real-time parallel applications consisting of tasks with varying execution time requirements. The tasks and inter-task precedence constraints (*i.e.*, communication and synchronization requirements) of the parallel applications are modeled as DAGs; vertices of a DAG represent computational tasks and edges represent communication tasks. To account for uncertainty in the time to complete computation and communication tasks, their processing time requirements are modeled as *independent random variables* with bounded minimum and maximum values. Independence of random variables implies that the observation of any particular value of one variable has no influence on the observed values of the other variables. The assumption of independent task execution requirements is justified because the causes of the variance in a task's execution time requirements are restricted to the effects of the advanced processor architecture; variances in execution times caused by data characteristics (*e.g.*, size and locality) and execution flows (*e.g.*, different conditional branches) are excluded. The use of bounded intervals for the values of completion times is justified because real-time tasks, by definition, are designed to reduce execution time jitter, and therefore, cannot have unbounded completion times.

The target hardware architecture is assumed to consist of a cluster of homogeneous processors with modern performance enhancing architectural features such as multiple levels of memory caches, speculative execution, branch prediction, and DMA that produce variations in completion times of computational tasks. Because these

features are provided in most current processors and their performance benefits are typically too significant to forego in applications and operating systems, the resulting variances in completion times must be accounted for in real-time schedules.

Communication between processors in the cluster is accomplished by passing messages. Message passing is a popular and well-known paradigm for providing communication between high-performance parallel processes [68]. Message passing operations between tasks scheduled to execute on the same processor are assumed to consume negligible time. The Real-Time Message Passing Interface (MPI) [140] standard provides buffer management mechanisms that can be exploited to reduce the cost of message transfers between unrelated processes (*i.e.*, copying buffers) through the use of shared memory buffer semantics.

By contrast, inter-node message passing requires the utilization of network resources. In packet-switched networks used in modern clusters, the queuing of packets multiplexed to the same output port is a significant cause of communication delays at the switches [58, 59]. Because this delay varies depending on the length of the queue and the time required to complete communication operations are also modeled as independent random variables. Furthermore, the scheduling algorithms developed in this dissertation assume that each cluster node is connected to the network via full-duplex links, modeled as a pair of simplex links. Each of these links is only capable of performing one non-preemptive communication operation at a time. The network fabric itself is assumed to be capable of carrying all offered network traffic without congestion.

Deterministic *list scheduling* (LS) [91] and *genetic list scheduling* (GLS) [72] techniques that have been used successfully in constructing static schedules for non-real-time parallel applications are extended in order to produce soft real-time schedules. Existing LS and GLS approaches assume that tasks have fixed execution time requirements and also typically ignore communication contention [91]. The stochastic scheduling algorithms developed and investigated in this dissertation assume tasks have variable execution time requirements and account for delays that occur when communication operations compete for processor-to-network links.

These stochastic scheduling algorithms use results from probability theory to compute the PDFs of an individual task's completion time from the task's execution time requirement PDF and the preceding tasks' completion time PDFs. In a schedule, preceding tasks of task J_i are those tasks specified by the precedence constraints in the DAG or other previously scheduled tasks using resources (*e.g.*, processors and communication links) required by J_i. Recall that in a non-preemptive schedule, J_i must wait for the required resources to become available before it can begin execution.

In deterministic scheduling approaches, the completion time of task J_i is computed by summing the execution time requirement of the task with J_i's starting time. The stochastic equivalent of this operation is convolution (*i.e.*, the completion time PDF of J_i, is computed by the convolution of J_i's starting time PDF and execution time requirement PDF) [18].

Task J_i can be allocated to an idle time slot in the schedule of a resource, provided that the slot begins at or after J_i can begin execution and has sufficient length to

accommodate J_i's execution time requirement. In deterministic approaches, the starting time of a candidate slot is given by the maximum completion times of all preceding tasks of J_i. Furthermore, the slot's ending time is given by the minimum starting time of all tasks previously allocated to the same resource scheduled to begin after the starting time of the candidate slot. Equivalent PDF operators for computing the minimum and maximum of sets of PDFs are developed in this dissertation.

Given the starting time and completion time PDFs of a candidate allocation slot in the schedule and the task's execution time PDF, the probability that the task completes before the slot ends is also computed. This probability is then compared against a "slot-fitting" probability threshold and the resource allocation is made in the slot the probability of timely completion is at least as large as the threshold value. The starting time PDF of each task is also used to determine the amount of time the task can be delayed in order to reduce the task completion jitter in the schedule.

The completion time PDF of the entire schedule is computed from the maximum of the PDFs of the *terminal tasks* in the schedule. Terminal tasks are those tasks that are not followed by other tasks in the schedule. The tradeoff between the probability of meeting the *end-to-end deadline* and the schedule length is computed from the completion time PDF of the schedule. End-to-end deadline is defined as the time relative to schedule start time by which all tasks in the schedule (*i.e.*, all vertices and edges in the corresponding DAG) must complete.

An important assumption of this research is that the task execution times are independent of each other. Therefore, the task start and completion time PDFs will not

be accurate if task execution times are dependent on each other. Recall that independence of task execution time requirements implies that the variations in tasks' execution time requirements are caused by the uncertainty induced by the advanced processor architecture features.

The PDF manipulation operations are computationally costly and the LS and GLS approaches evaluate the suitability of several potential time slots for allocation to each task in the DAG. Therefore, the construction of stochastic schedules can take a long time to complete when PDF manipulations are used at every step of the scheduling algorithm. In order to reduce the amount of time taken to compute schedules, an alternative approach to scheduling is also investigated. Under this approach, a fixed estimate of the execution time requirements of each task is used instead of the tasks' execution time PDF to construct an initial schedule. The task-resource allocations and task sequences from the initial schedule are used to construct the final stochastic schedule using task execution time PDFs.

The approach of using PDF manipulations at every scheduling step is designated as the *exact method* and the approach of using estimates for initial scheduling is designated as the *estimate method* in this dissertation. The estimate method has the potential for substantially reducing computation time compared to the exact method because most of the scheduling decisions have already been made by the time PDF manipulation operations are used in the estimate method. Consequently, the total number of PDF computation is significantly reduced in the algorithms using the estimate method. However, the estimate method algorithms are unable to exploit the slot-fitting threshold.

Therefore, the performance of the two approaches in terms of schedule lengths is compared in this dissertation.

1.7 Experimental Plan

In order to validate the hypothesis and in order to test the scheduling approaches developed in this research, schedules for a number of sample DAGs are constructed using these approaches. The characteristics of these schedules (*e.g.*, schedule length, probability of meeting end-to-end deadlines, task completion jitter, resource utilization, and time to construct the schedule) are analyzed and compared with each other.

In order to examine the properties of schedules for a wide range of problems, sample DAGs with a variety of characteristics are generated. The primary distinguishing characteristic of a DAG is its overall structure type, determined by the connectivity of the vertices and edges. DAGs with structures commonly observed in typical parallel applications (*e.g.*, the "fork-join" structure of client/server applications, and parallel FFT structure) are generated. DAGs with random acyclic structures are also generated in order to represent applications that have irregular, but known, computation and communication patterns. Task and edge weights in these DAGs are modeled as random variables with a variety of distributions (*e.g.*, *exponential, beta*, and *random* distributions).

Other DAG characteristics that are also varied are the *computation-to-communication* ratio (CCR) (*i.e.*, the ratio of average vertex weight to average edge weight) and the size of the DAG in terms of the total number of vertices and edges.

For each sample DAG, schedules are constructed and analyzed using a number of different approaches with varying control parameters. In particular, schedules resulting from the LS approach using the estimate method and the exact method for PDF computation are compared with each other. The extent to which reducing the required probability of completing within the deadline reduces the schedule's length is investigated for the shortest schedule for each DAG. The effect of varying the jitter control parameter on resource utilization and the tradeoff between QoS and schedule lengths is also investigated.

A similar series of scheduling experiments and analysis is also performed for the FFT structured DAGs using the GLS approach. The GLS approach is also compared with the LS approach in terms of overall schedule lengths and time taken to compute the schedules.

1.8 Contributions of this Dissertation

A primary contribution of this research is the generalization and extension of the traditional LS and GLS approaches in order to schedule soft real-time tasks with varying task execution time requirements. Traditional LS and GLS approaches assume that tasks have fixed execution time requirements. Conversely, the execution time requirements of soft real-time tasks are typically modeled as PDFs. However, a fixed execution time for a task is essentially a special case of a PDF in that the single fixed execution time occurs with 100% probability. In LS, the execution time requirements are manipulated using addition, subtraction, minimum, and maximum operations in order to compute task start and completion times. This dissertation uses convolution to sum PDFs and develops new

operations for computing the difference, minimum, and maximum of PDFs required in LS and GLS. The use of these PDF operations for LS and GLS is also a new contribution of this dissertation (convolution has been used extensively in previous research for performing preemptive periodic scheduling [18, 61, 152]).

Another significant contribution of this research is the empirical demonstration of the veracity of the hypothesis. Experimental results clearly show that through the use of PDF manipulations, the end-to-end completion time PDF of a schedule can be computed accurately. This PDF can then be used to systematically reduce the length of the schedule if a less than 100% probability of meeting end-to-end deadlines is acceptable. The results show that a small reduction in the required end-to-end probability of meeting deadlines can result in a significant reduction of schedule length as compared to the schedule that assumes the WCET requirements.

New heuristic parameters that exploit the task start and completion time PDFs to control the placement of tasks during schedule construction and to control task completion jitter are also developed and investigated in this research. Results show that the use of the slot-fitting threshold heuristic can significantly reduce schedule lengths for many DAGs. Results also show that the use of the jitter control parameter effectively reduces the task completion jitter in schedules, albeit at the cost of reduced ability to trade probability of meeting end-to-end deadlines for schedule length.

This dissertation provides the PDF manipulation algebra and scheduling approaches that can be combined to construct schedules for soft real-time parallel applications consisting of tasks with varying execution time requirements. Schedules for

pragmatic soft real-time systems, where QoS requirements must be balanced with performance, resource utilization, and jitter, can be constructed using the techniques developed in this dissertation.

1.9 Organization of the Dissertation

The remainder of this dissertation is organized as follows: Chapter 2 introduces the fundamental concepts, terminology, and state of the art in the research, design, and engineering of real-time and non-real-time scheduling, real-time communication, and real-time operating systems design. Chapter 3 presents the model for the parallel soft real-time applications, the model for the real-time hardware, and the scheduling approaches addressed in this dissertation. Chapter 4 describes the experimental setup used to validate the hypothesis. Chapter 5 presents and analyzes the result of the experiments. Chapter 6 concludes with a summary of research results, contributions, and a description of potential extensions to this research.

CHAPTER II

LITERATURE REVIEW

This chapter introduces basic terminology, concepts, and properties of real-time systems and real-time scheduling. This chapter also summarizes related work in scheduling, deterministic real-time scheduling, and probabilistic real-time scheduling. Selected real-time operating systems and real-time communication techniques are also surveyed here. This chapter ends with a discussion of limitations of existing real-time scheduling research.

2.1 Introduction

A number of advances have been made in the fields of scheduling, operating system kernels, communication, and design and analysis techniques for real-time systems over the last two decades and are summarized in this chapter. Theoretical studies in scheduling of tasks under a variety of constraints have produced significant results describing the complexity of the scheduling problems [24, 100, 153]. These complexity results play an important role in the selection of scheduling algorithms that best match the problem at hand. Theoretical analysis of online dynamic scheduling has produced important results such as bounds on worst-case execution time (WCET), and achievable resource utilization [26, 61, 78, 96, 98, 99, 101, 158, 109].

Applied research in scheduling has resulted in techniques for generating static schedules that provide hard real-time guarantees [26, 118, 158]. Practical extensions to dynamic scheduling algorithms have been proposed in order to account for shared resources, in addition to the traditional focus on deadlines and periods [26, 136]. A number of practical techniques addressing the problems of admission control in real-time system have also been studied [10, 15, 58, 59, 95, 159, 166].

Research in the area of real-time operating systems (RTOS) kernels has resulted in a number of Unix-like kernels that strive to minimize overheads by limiting context switch time, reducing interrupt latency, and pre-allocating system resources to tasks. These kernels also provide priority-oriented task dispatching, timeout mechanisms, and real-time clocks [5, 16, 95, 118, 123, 124, 131, 151].

A variety of theoretical models for real-time system area networks (SANs) and local area networks (LANs) have been proposed that provide connection-oriented and connectionless communication facilities [112]. Additionally, a number of practical real-time protocols that reduce or avoid collisions in order to provide deterministic communication facilities appropriate for distributed real-time systems have been studied [4, 33, 39, 49, 70, 76, 82, 85, 88, 89, 102, 122, 150, 154].

An accurate understanding of the temporal behavior and resource access patterns of real-time tasks is critical in the construction of real-time schedules and many scheduling algorithms rely on the knowledge of WCET of tasks. Therefore, a number of techniques for determining WCET of tasks, based on the timing analysis of instructions

and control flow in real-time tasks, have been studied and reported in the literature [71, 103, 105, 106, 110].

This chapter surveys current-practice and research in scheduling, RTOS kernel design, communication protocols, and timing analysis techniques that form the foundation of this dissertation.

2.2 Properties of Real-time Tasks

This section presents a unified notation derived from the literature for representing properties such as constraints, requirements, and relations of real-time tasks. A real-time system is typically composed from a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n periodic, *sporadic*, and aperiodic tasks. A sporadic task is a periodic task that may not be triggered by the systems environment to execute in every period. The k^{th} instance of a recurring (*i.e.*, periodic or sporadic) task J_i is represented by J_i^k where $1 \leq k < \infty$. Because aperiodic tasks are modeled to have only a single instance (*i.e.*, $k = 1$), the superscript designating the instance is typically omitted. Following are properties commonly used in the literature to characterize instances of real-time tasks [24, 26, 96, 158]:

- *Execution time* is the time required by task J_i to complete. Compute time is modeled to be invariant across all instances J_i^k and is denoted by w_i. Typically, w_i represents the WCET of J_i.
- *Period* is the length of the intervals between successive activations of task J_i and is denoted by T_i. Period is relevant only for recurring tasks.

- *Frequency* is the maximum frequency of activations of task J_i the system is expected to support and is denoted by υ_i. Frequency is relevant only for sporadic tasks. The period of a sporadic task can be determined from its frequency using the relation $T_i = 1/\upsilon_i$.
- *Release time* (synonymous with *ready time* and *arrival time*) is the time relative to the schedule's start time when a task instance becomes available for execution. Release time is denoted by r_i^k. For recurring tasks, the release time is given by the following:

$$r_i^k = \phi_i + (k-1)T_i. \tag{2.1}$$

- *Phase* is the time relative to the schedule's start time when the first instance of a recurring task J_i is released and is denoted by ϕ_i. Note that $\phi_i \equiv r_i^1$.
- *Relative deadline* is the amount of time relative to release time within which task instances must complete. The relative deadline is invariant across all instances of the same task and is denoted by D_i.
- *Deadline* is the absolute time by which a task instance must complete in order to meet real-time performance requirements and is given by the following:

$$d_i^k = r_i^k + D_i. \tag{2.2}$$

- *Start time* is the absolute time when the task instance begins execution and is denoted by s_i^k.
- *Finish time* is the absolute time when the task completes and is denoted by f_i^k.
- *Lateness* is the difference between a task instance's deadline and finish times. Lateness is computed as follows:

$$L_i^k = f_i^k - d_i^k. \tag{2.3}$$

Negative lateness results when the instance completes before its deadline.

- *Tardiness* is the time by which a task's instance exceeds its deadline and is computed as follows:

$$E_i^k = max(0, L_i^k). \tag{2.4}$$

- *Laxity* (also known as *slack time*) is the amount of time a task can exceed its compute time before missing its deadline and is computed as follows:

$$X_i^k = d_i^k - r_i^k - C_i^k. \tag{2.5}$$

- *Precedence*, denoted by the non-reflexive binary relation $J_a \prec J_b$, specifies that task instance J_b^q cannot begin execution until task instance J_a^p has completed where the pair $(p, q) \in \{(x, y) \mid \text{if } T_a = cT_b \text{ then } y = cx \text{ or if } T_b = cT_a \text{ then } x = cy \text{ and } c \geq 1\}$.
- *Exclusion*, denoted by the binary relation $J_a \prec\succ J_b$, means that instances of tasks J_a and J_b cannot preempt each other. However, other tasks are not restricted from preempting either J_a or J_b (unless they also have an exclusion relationship with either J_a or J_b). Note that the exclusion relationship is only relevant in systems allowing preemption.
- *Preemption* is a Boolean property that indicates whether or not the task can be preempted by another task. A task's preemption property is invariant across all instances.
- *Criticality* is a Boolean property that indicates whether or not timely execution of the task is essential for system correctness. Typically, hard real-time tasks are considered

critical and are given preference over non-critical (or soft real-time) tasks under overload conditions. A task's criticalness is typically invariant across all instances.

- *Priority* is a numerical quantity describing the importance of this task relative to other tasks and is denoted by ρ_i. A task's priority is typically invariant across all instances. However, the system may temporarily adjust the priority of any task instance in order to adjust schedulability and hence to meet system objectives.

2.3 Scheduling Real-Time Tasks

Task scheduling is the problem of assigning resources to tasks over time intervals such that all constraints are satisfied and some evaluation, or quality, criterion is optimized. Scheduling algorithms employed in traditional interactive operating systems strive to minimize response and turnaround times. Essentially, the operating system (OS) strives to maximize the number of tasks completed in a given unit of time and to maximize resource utilization [146]. Objective functions typically used to evaluate the quality of scheduling in operating systems given a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n aperiodic (*i.e.*, non-recurring) jobs with arbitrary arrival and compute times are as follows [24, 26, 146]:

- Average response time is computed as follows:

$$\bar{t}_r = \frac{1}{n}\sum_{i=1}^{n}\left(f_i - r_i\right). \tag{2.6}$$

- Total completion (or flow) time is computed as follows:

$$t_c = \sum_{i=1}^{n} (f_i) . \tag{2.7}$$

- Weighted sum of completion (flow) times is computed as follows:

$$t_\omega = \sum_{i=1}^{n} \omega_i f_i , \tag{2.8}$$

where $\omega_i \in \Re$.

- Makespan (*i.e.*, schedule length) is computed as follows:

$$t_m = \max_i (f_i) . \tag{2.9}$$

- Throughput is computed as

$$\eta = \frac{\sum_{i=1}^{n} w_i}{\max_i (f_i)} . \tag{2.10}$$

By contrast, in real-time scheduling, minimizing maximum lateness or minimizing the number of late tasks is more important than reducing average response times or maximizing resource utilization. Furthermore, real-time systems are typically comprised of periodic and sporadic tasks combined with aperiodic and *best-effort* tasks. Best-effort tasks are those that do not have real-time requirements and are executed whenever system resources are not allocated to real-time tasks. Best effort tasks may also be periodic, sporadic, and aperiodic. A primary objective of real-time schedules is to execute best-effort tasks in a manner that minimizes their impact on the predictability and timely completion of the real-time system components.

When performing schedulability analysis of a set of periodic real-time tasks, the set's *processor utilization* factor plays an important role. Processor utilization is the fraction of time spent executing the tasks. For periodic tasks in a uniprocessor environment, processor utilization is computed as follows:

$$U = \sum_{i=1}^{n} \frac{w_i}{T_i}. \tag{2.11}$$

Clearly, a task set with $U > 1$ cannot be successfully scheduled using any algorithm. The *breakdown utilization*, U_A^*, is the upper bound on the utilization factor within which a given periodic scheduling algorithm A can guarantee a feasible schedule for an arbitrary set $\mathcal{J}$ of periodic tasks (*i.e.*, $U < U_A^*$ guarantees that A can successfully schedule all tasks in $\mathcal{J}$). If $U_A^* < U \leq 1$, then A may fail to construct a feasible schedule, depending on the timing characteristics of the various tasks in J.

2.3.1 Deterministic Scheduling

Many real-time applications require the flexibility to schedule dynamic workloads wherein arrival and computation times of the tasks are variable and cannot be assumed to be deterministic at system design time. Such systems are said to be *event-driven* because tasks are activated in response to environmental stimuli [26, 31]. Event-driven systems typically utilize dynamic scheduling algorithms during runtime whenever new tasks arrive or existing tasks terminate in order to adapt to variations in workload. When a new

task arrives, the scheduling algorithm performs schedulability analysis, also known as an *admission test* [26, 31], to determine whether or not the task will be *admitted* (*i.e.*, accepted for execution). If the algorithm can construct a feasible schedule from the newly arrived and all previously admitted tasks, the new task is also admitted. Otherwise, the new task is rejected.

A number of dynamic algorithms based on a variety of heuristics have been reported in the literature. Jackson's Earliest Due Date (EDD) algorithm minimizes the maximum lateness of a set of independent tasks with identical arrival times by executing the tasks in order of non-decreasing absolute deadlines [78]. The complexity of EDD is $\Theta(n\log_2(n))$ because the tasks must be sorted by deadlines. Horn's Earliest Deadline First (EDF) algorithm minimizes the maximum lateness of a set of independent tasks with arbitrary arrival times [73]. Whenever a new task arrives or an executing task completes, EDF selects the ready task with the earliest deadline for execution first. EDF performs preemptive scheduling and has a complexity of $O(n^2)$. Furthermore, EDF is guaranteed to find a feasible schedule for a task set J if such a schedule exists [43].

Rate Monotonic Scheduling (RMS) is used to schedule periodic tasks with deadlines equal to task periods [109]. RMS assigns higher priority to tasks with shorter periods than to tasks with longer periods (*i.e.*, frequently occurring tasks have higher priority). For RMS, the breakdown utilization factor for a set of n arbitrary tasks is given by $U^*_{RMS} = n(2^{1/n} - 1)$ and for large n, $U^*_{RMS} = \log_e(2) \approx 69\%$. Results from stochastic simulation studies show that when RMS is applied to large uniformly distributed random

task sets, U^*_{RMS} is dependent on task periods, instead of task computation times, and converges to 88% [98].

Deadline Monotonic Scheduling (DMS) is an extension to RMS that considers sets of tasks with deadlines shorter than their respective periods. DMS grants higher priority to tasks with the shorter relative deadlines than to tasks with longer relative deadlines [101]. For task $J_i \in \mathcal{J}$ to be successfully scheduled by DMS, the sum of C_i and the interference caused by higher priority tasks must be less than or equal to D_i. This sum is called the *response time* of J_i and is denoted by R_i. Therefore, if tasks in $\mathcal{J}$ are sorted in non-decreasing relative deadline order, then task J_i is schedulable if and only if $R_i \leq D_i$. The response time is computed as follows:

$$R_i = w_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil w_k \,. \tag{2.1}$$

Because R_i appears on both sides of the equation, an iterative approach is used to determine the feasibility of scheduling J_i [11].

In situations where hard real-time periodic and soft real-time aperiodic tasks must be scheduled together, the periodic tasks are scheduled using the methods described above and the aperiodic tasks are executed when there are no periodic tasks to execute. However, this background scheduling of aperiodic tasks can significantly extend their response times when the periodic load is high.

In order to reduce the response times of aperiodic tasks or to guarantee execution of hard real-time sporadic or aperiodic tasks, *servers* are used, where a server in this context is a periodic task that is used to service sporadic and aperiodic requests. A

Polling Server (PS) [99] is activated at regular intervals of period T_s and has *capacity* (*i.e.*, compute time) of C_s. (Note that $T_s > C_s$.) After activation, the PS handles any pending aperiodic and sporadic requests up to a maximum time of C_s. If no aperiodic or sporadic requests are pending, the PS is suspended and the remaining PS capacity is used to execute periodic tasks.

Deferrable Server (DS) [99] scheduling is based on the observation that completing a hard real-time task earlier than necessary does not improve its value. In DS scheduling, if no aperiodic tasks are pending, a periodic task is scheduled. However, the *capacity* (*i.e.*, the reserved execution time) of DS is preserved. Therefore, when an aperiodic task arrives, the executing hard real-time periodic task is preempted to execute the DS as long as DS capacity remains.

In those periodic real-time systems where there is no advantage to be gained from completing tasks earlier than their respective deadlines, the *Slack Stealing* algorithm [97] can be used instead of DS. The *Slack Stealing* algorithm uses the slack from periodic tasks to service aperiodic tasks. Slack at time t for a periodic task J_i^k, is given by the following equation:

$$slack_i^k(t) = d_i^k - t - c_i^k(t), \tag{2.2}$$

where $c_i^k(t)$ is the computation time for task J_i^k that remains at time t. When no aperiodic tasks are pending, RMS is used for scheduling the periodic tasks. Slack Stealing has better response times for aperiodic tasks as compared to the PS and DS algorithms.

The *Total Bandwidth Server* (TBS) [145] also strives to reduce response times for aperiodic tasks in an EDF periodic scheduling environment. When an aperiodic task arrives, it is assigned the following deadline:

$$d_i = \max(r_i, d_{i-1}) + \frac{w_i}{U_s}, \tag{2.3}$$

where r_i and w_i are the release time and execution time requirements for the i^{th} aperiodic task, respectively; and U_s is the percent of total system utilization capacity that can be allocated by the system designers for servicing aperiodic tasks.

Preemption is the key to successful dynamic scheduling because the scheduler can interrupt the currently executing task in order to accommodate more critical tasks. Figure 2.1 illustrates the scenario where task J_b arrives after and overlaps J_a (*i.e.*, $r_a < r_b < r_a + w_a$), and J_a has a sufficiently long deadline to accommodate interference from J_b (*i.e.*, $w_b + w_a < D_a$). When J_a arrives, the online dynamic scheduling algorithm has no knowledge of J_b and begins executing J_a. In the case where preemption is permitted, the scheduler interrupts J_a in order to execute J_b and both tasks complete before their respective deadlines expire. In the case where preemption is not permitted, J_b must wait for J_a to complete and consequently J_b misses its deadline.

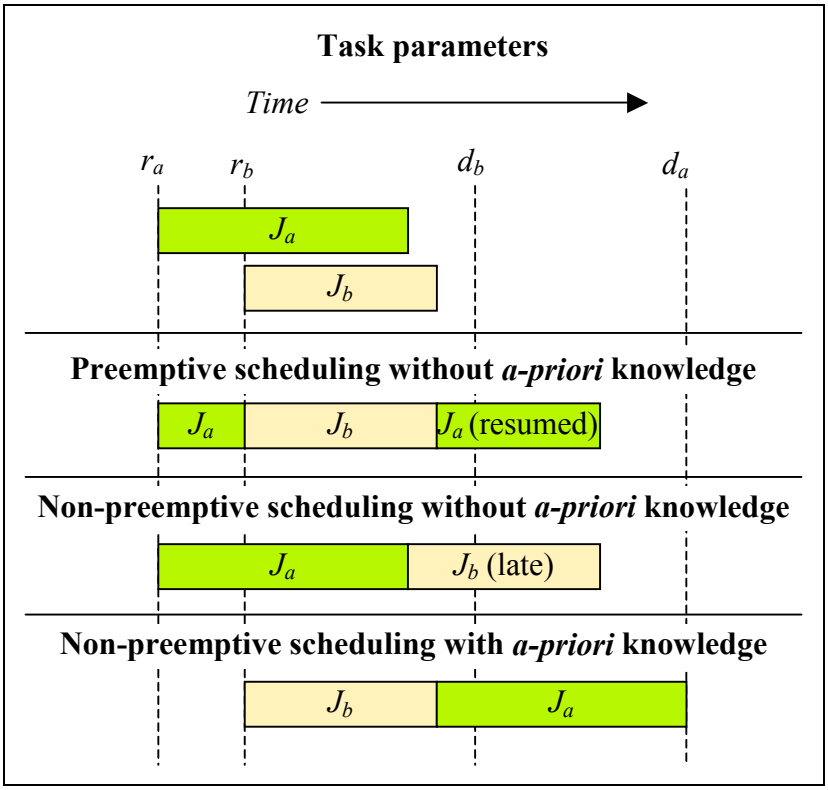


Figure 2.1 Gantt Chart for Preemptive and Non-preemptive Scheduling

When the scheduler has prior knowledge of the timing characteristics of J_a and J_b, it can delay the activation of J_a until J_b has completed, thereby ensuring that both tasks meet their deadlines even when preemption is disallowed. Static scheduling algorithms use prior knowledge of a fixed set of tasks to compute optimal schedules by enumerating and implicitly or explicitly evaluating all possible feasible scheduling alternatives. However, a variety of scheduling problems are known to be NP-hard in general [100, 153]. Therefore, static scheduling algorithms are typically executed offline.

The literature from Operations Research is replete with techniques for solving scheduling problems. Of these, the job-shop scheduling problem has received much

attention because of its general applicability and complexity. One approach to solving scheduling problems is to reduce the problems to combinatorial optimization problems that can be solved using mixed integer linear programming [113, 149]. In linear programming, the scheduling problem to be solved is restated in terms of minimizing (or maximizing) a linear function subject to linear constraint functions. However, the number of variables required to solve practical job-shop problems grows exponentially and success with linear programming has been restricted to small problems [20, 79].

Other approaches to solving job-shop problems involve utilizing variants of branch-and-bound (BB) techniques [25, 114, 158]. In BB, the search is characterized by a tree where interior nodes represent partial solutions and leaf nodes represent schedules. In the "branch" step, the partial schedules at parent nodes are refined. In the "bound" step, nodes are discarded if the estimated lower bound on their cost is larger than the currently known upper bound on the optimal schedule cost. The optimal upper bound decreases monotonically as better schedules are found during the search. In order to increase the search space pruning effectiveness of the bounding step, the lower bound on the cost of the node must be as tight as possible. An *admissible* heuristic is one that does not overestimate the schedule cost at a node. The use of admissible heuristics ensures that BB will find an optimal schedule, whereas an overestimated cost can potentially cause a sub-tree containing an optimal solution to be incorrectly pruned from the search. However, a heuristic that underestimates cost by a significant margin diminishes the pruning ability of BB, resulting in longer search times.

Instead of focusing on producing optimal schedules, a number of researchers have proposed "approximation algorithms" that produce schedules with lengths within guaranteed bounds relative to the optimal schedule length [35, 80, 138]. Similarly, good results have been obtained through the use of Lagrangian Multipliers to relax constraints in linear programming formulations of scheduling problems [36, 50, 51]. Techniques based on splitting a large problem into smaller sub-problems and obtaining optimal solutions to the smaller linear programming problems has also shown limited success [8, 137].

Many search algorithms from AI (*e.g.*, variants of constraint satisfaction techniques [32, 52, 116], simulated annealing [94, 29], and genetic algorithms [19, 115, 117]) use imprecise heuristics to prune large unpromising portions of the search space. These approaches avoid examining the entire solution space (either explicitly or implicitly), and therefore, sacrifice optimality in order to find high-quality schedules relatively quickly. These approaches are described next.

Solutions to constraint satisfaction problems require the assignment of values to a set of variables where the value assignments are subject to a set of constraints. In order to find near-optimal schedules quickly, the requirements of minimizing makespan and meeting all constraints are relaxed [32, 52, 116].

Simulated annealing (SA) [94, 29] is a Monte Carlo approximation technique used to obtain near-optimal solutions to large combinatorial optimization problems. It is based upon the analogy between solving optimization problems and the physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure.

The quality of the schedules produced by SA depends on the "energy" of the initial solution representing the molten state, the "annealing schedule", and the "temperature" values at each stage of the annealing schedule. In SA, the value of the objective function is analogous to energy. The probability with which a random change to the solution producing a higher energy solution is accepted is analogous to temperature. The number of iterations at each temperature level and amount by which the temperature is lowered at each stage is analogous to an annealing schedule. Because SA utilizes little problem specific knowledge, the number of iterations required to solve large scheduling problems is large. Therefore, a number of techniques, (*e.g.*, in [44] and [130]) for accelerating the SA process have been proposed.

Genetic algorithms (GAs) search large, multi-dimensional combinatorial spaces by emulating the evolutionary processes found in nature [19, 115, 117]. Based on their relative *fitness* (*i.e.*, value of the optimization criteria), individuals are selected from a population of potential solutions to contribute their characteristics to the next generation via a set of recombination operations (*e.g.*, crossover and mutation). This process is repeated until the solution is found. GA implementations that maintain several populations isolated form each other to a certain degree are more resistant to premature convergence to local optima as compared to implementations with a single implementation [27, 107, 162]. Furthermore, GAs are particularly attractive scheduling approaches because they are easily parallelized. However, GAs' proclivity to converge to local optima limits their applicability to schedule construction. Therefore, basic GA

techniques are extended through the use of complex data structures and evolutionary operators. Such extensions are referred to as evolutionary programming (EP) [115].

2.3.2 Stochastic Scheduling

In many soft real-time applications, tasks have highly variable execution time requirements. For such applications, meeting deadlines with some minimum guaranteed probability is required (*i.e.*, missing occasional deadlines is acceptable). Therefore, these applications do not require schedules that plan for the worst-case scenario. Instead, schedules for such applications improve system performance and utilization by only guaranteeing that deadlines will be met with a given minimum probability.

Statistical Rate Monotonic Scheduling (SRMS) [10] extends the analysis of RMS for scheduling periodic tasks with variable runtimes and statistical real-time guarantees. As in RMS, SRMS assigns a fixed priority to each task and preemptively schedules tasks based on their priorities. Higher priority is assigned to tasks with shorter periods. Variability in a task's execution time is accounted for by allocating a time budget for successive instances of the task. The scheduler ensures that each task is granted resources according to the tasks' time budget on average. An admission control procedure ensures that only those tasks that are not prevented from meeting their deadlines by higher priority tasks and having sufficient budgets are scheduled for execution.

The Basic SRMS algorithm applies to *harmonic* task sets [10]. A task set is considered harmonic if the periods for all tasks are integral multiples of the periods of all

tasks with smaller periods. For a harmonic task set, the probability that an instance of task J_i will be admitted is given by the following "QoS" function:

$$QoS(J_i) = \frac{\hat{T}_i}{\hat{T}_{i+1}} \sum_{k=1}^{\frac{\hat{T}_{i+1}}{\hat{T}_i}} P(S_{i,k} = 1), \tag{2.4}$$

where $\hat{T}_{i+1}$ is the *superperiod* of J_i, and $P(S_{i,k} = 1)$ is the probability that the task instance J_i will be admitted in the k^{th} phase of the superperiod. A superperiod of a task is the period of the next lower priority task. $P(S_{i,k} = 1)$ is computed by summing the probabilities of all possible combinations of task instances of J_i being accepted or rejected in the prior k-1 phases.

Probabilistic Time-Demand Analysis (PTDA) [152] computes the probability that a task instance J_i^k will complete within its deadline. This is done by computing the lower bound, c_i, on the total amount of time required to complete J_i^k and all other higher priority task instances that are released in the interval $[r_i^k, r_i^k + t)$, for any $t > 0$. When the computational requirements for tasks are variable, c_i itself is variable. Assume that $CDF_i(x)$ is the cumulative probability density function [45] of c_i (*i.e.*, the probability that $c_i \leq x$), then the probability that J_i^k completes at or before its deadline is given by $CDF_i(D_i)$. $CDF_i(x)$ is computed by convolving the PDFs of the computation time requirements of J_i^k and all other higher priority task instances that are released before J_i^k completes. In PTDA, the relative deadline of a task is assumed to be less than or equal to the task's period.

Stochastic Time Demand Analysis (STDA) [61] extends PTDA to include tasks whose relative deadlines are greater than their periods. STDA also accounts for the time that tasks can block each other while accessing shared resources. Note that a higher priority task cannot preempt a lower priority task that is in a *critical section*. A critical section is a section of code in which a task needs exclusive access to shared resources.

STDA is also applied to distributed systems by assuming that a periodic task is composed of a chain of "*subjobs*" with each subjob executing sequentially on a different processor [61]. Static priority assignment is used to determine subjob priorities. The period of each subjobs is assumed to be the same as the period of the task and the inter-release time for two consecutive instances of a subjob is assumed to be at least as long as the task's period.

Abeni and Buttazzo [1, 2] present an approach for constructing schedules for periodic tasks with variable execution time requirements and variable inter-arrival times by using a *bandwidth reservation* strategy. Under this strategy, each task is assigned to a dedicated, periodic, *Constant Bandwidth Server* (CBS) that guarantees that the task will be executed for a pre-assigned fraction of the total CPU bandwidth. The completion of any task that requires more time than is available in the current period of its CBS, is delayed until the next period of its CBS. This mechanism isolates tasks and prevents tasks from delaying the completion of other tasks and the schedules are guaranteed to meet deadlines with a given probability. Schedulability analyses for the following two scenarios are also presented: 1) task sets with variable execution time requirements and

constant inter-arrival times; and 2) task sets with constant execution time requirements and variable inter-arrival times.

The work of Diaz et al. [46] extends the analytical approaches introduced in STDA and PTDA to include both fixed priority (*e.g.*, RM and DM), and dynamic priority (*e.g.*, EDF) periodic real-time systems. Furthermore, analytical approach is also developed to include periodic tasks with "arbitrary relative deadlines (including relative deadlines greater then the periods) and arbitrary execution time distributions."

Ryu and Kim present techniques for scheduling with statistical deadline guarantees, a mixed set of periodic and aperiodic tasks [129]. In their scheduling approach, each task is assigned a fixed amount of processor time, called the *effective execution time*. A task that exceeds its effective execution time or exceeds it deadline is discarded from the system. This simple strategy enables the scheduler to efficiently control the probabilities of the tasks missing their deadlines. Essentially, this strategy isolates tasks from each other and a single misbehaving task cannot adversely affect the probability of another task from completing in time.

The effective execution time, $\bar{e}$, of a task J_i is computed from the following equation:

$$\int_0^{\bar{e}_i} \pi_{wi}(x)dx = 1 - \varepsilon, \tag{2.5}$$

where $\pi_{wi}(x)$ is the probability distribution function of the completion time of task J_i, and ε is the required deadline miss probability. The effective execution time is then used in a deterministic admission control technique, *utilization demand analysis* (UDA), to

construct schedules. Under UDA, the utilization demand, $u_i(t)$, of task J_i at time t is given by the following equation:

$$u_i(x) = \frac{\bar{e}_i + \sum_{K \in Q(t, J_i)} \bar{e}_K}{d_i - t}, \tag{2.6}$$

where $Q(t, J_i)$ is the set of tasks with higher priority than J_i. Recall that d_i is J_i's deadline. Given this definition for utilization demand, a set of aperiodic tasks is schedulable if and only if the maximum utilization demand of the set is less than or equal to 1.0. Furthermore, Ryu and Kim also show that a set of periodic and aperiodic tasks is schedulable by an EDF scheduler if the sum of the utilization of the periodic tasks and the maximum utilization demand of the aperiodic tasks is less than or equal to 1.0.

Because scheduling using the absolute WCET for tasks is overly pessimistic, a number of research efforts have focused on constructing a probabilistic model of the worst-case behavior of the system. A sampling of such efforts is summarized below.

Bernat, Colin, and Petters [18] introduce techniques for constructing the "execution profiles" of tasks by measuring the variable execution time for "basic blocks" of program code of which the tasks are composed. Probabilistic analysis techniques are used to combine the variable worst-case behaviors of the blocks to obtain the overall probabilistic WCET estimates of each task in a soft real-time system. In particular, techniques for combining the probability distributions functions of different basic blocks that are executed in sequence, in an "if-then-else" structure, and in loops are developed and presented. Two separate sets of techniques are developed in order to cover the cases

where the execution times of the basic blocks are independent of each other and are dependent on each other.

In a simple approach for estimating the WCET of a real-time task, the execution time of a task is measured on a real processor and the largest execution time from a number of repeated observations is used as the WCET. However, the probing codes used to take the measurements can introduce variations that will not occur in the final real-time system. Furthermore, the maximum execution time determined from a sample of executions is not guaranteed to be greater than or equal to the WCET that may be observed in a deployed system. In order to account for the lack WCET guarantees estimated from experimentation, Edgar and Burns present an approach for using the *confidence level* of the estimated WCET for deriving the confidence level of the schedule that uses the estimated WCET [47]. The confidence level of an assertion "the WCET of task *J* is 200ms" is the probability that the assertion is indeed true for all possible executions of task *J*. Note that this is subtly different from the probability that execution time for task *J* is *2*00ms.

Edgar and Burns also present techniques for deriving the confidence level of WCET for tasks from sample observations and for trading the confidence level for improving system performance (*i.e.*, reducing the WCET of tasks – and their associated confidence level – in order to achieve schedulability) [47]. They show that a real-time schedule has a confidence level of at least $(1 - \varepsilon)$ of completing within the deadline as long as each task in the schedule has WCET with confidence level $(1 - \varepsilon)$ and the sum of the following probabilities is greater than or equal to 1.0:

1. a guaranteed schedule can be constructed with a confidence level of $(1 - \varepsilon)$ given that the WCET of the first $(n - 1)$ tasks are overestimated and the WCET of the n^{th} task is underestimated, and
2. a guaranteed schedule can be constructed with a confidence level of $(1 - \varepsilon)$ given that the WCET of the first $(n - 1)$ tasks are underestimated and the WCET of the n^{th} task is overestimated.

2.4 Real-Time Operating Systems

There are a number of real-time operating systems offered by a variety of commercial vendors and research groups. The survey in this section is restricted to a few well-known real-time operating systems (see [62] for a more extensive survey).

Commercial operating systems such as pSOS [155], VxWorks [156], and LynxOS [111] typically target embedded systems and are designed to be easily portable and support a variety of embedded hardware systems. These operating systems focus on providing low interrupt latency combined with low context switching overheads in order to provide real-time response to external events. Dynamic online scheduling policies such as FIFO, round robin, RMS, and EDF are provided; calendar-based scheduling is generally not supported. Real-time processing is provided through fine user-control of thread priorities, priority inheritance mechanisms, and preemptive thread scheduling.

In the Spring [118] system, designed for multiprocessor real-time computing, tasks are classified as being critical, essential, and non-essential. Sufficient resources are reserved by the Spring scheduler so as to guarantee that critical tasks complete within

their deadlines. This scheduler uses WCET of tasks, task deadlines, and resource requirements to construct non-preemptive schedules. Executing tasks are protected from external interrupts through the use of dedicated I/O processors and the operating system itself. Another processor is dedicated to performing administrative and scheduling functions. The Spring kernel also includes facilities for explicit control of translation lookaside buffer (TLB) contents in order to reduce the unpredictability caused by TLB misses [124]. The overhead introduced by this explicit TLB management adversely affects context-switching time.

In the Maruti [131] system, resources are reserved, *a priori*, to ensure timely task execution. Maruti has evolved from using static calendars to using parametric scheduling in order to increase scheduling flexibility when task execution times are variable [132]. In Maruti's parametric scheduling, a task's start time is determined by the execution times of previous tasks. In order to perform timing analysis of executable modules at compile time, module development in Maruti is performed using the Maruti Programming Language (MPL), a subset of ANSI C. MPL disallows features such as the `goto` keyword, unbounded loop variables, and recursion. The Maruti Configuration Language (MCL) is used to compose applications from individual modules.

The HARTIK [95] system uses EDF scheduling to execute real-time periodic and aperiodic tasks. Real-time task are further classified as being critical and non-critical. Critical tasks are given priority over non-critical tasks. Non real-time tasks are executed when no real-time tasks are active. Admission control based on WCET is used to ensure

schedule feasibility. HARTIK also uses the Stack Resource Policy (SRP) [14] to bound priority inversion and blocking time, and to prevent deadlocks.

RT-Mach [151] provides real-time services to periodic and aperiodic hard and soft real-time tasks through the use of round-robin fixed-priority scheduling policies. Real-time threads are allocated to processors *a priori* and do not migrate at runtime. An executing thread whose scheduling quantum has expired is preempted by another thread with equal priority. A higher priority thread always preempts a lower priority thread. Schedulability analysis is performed in order to determine and reserve the processing capacity required by the hard real-time periodic and sporadic tasks. Remaining processing capacity is allocated to soft real-time tasks.

A number of real-time executives based on Linux have been developed. The primary difficulty in using Linux as a real-time OS is that the non-reentrant kernel is locked during kernel calls and interrupts may be arbitrarily disabled. This can cause unbounded delays in task invocations and exacerbates interrupt latencies. RT-Linux [16] inserts a small-footprint real-time kernel between Linux and the hardware. The RT-Linux kernel executes the Linux kernel (and consequently Linux processes) as non real-time tasks and real-time tasks are executed in RT-Linux kernel space. RT-Linux intercepts interrupts, intercepts interrupt enabling and disabling kernel functions, and forwards interrupts to the Linux interrupt handlers only when Linux does not have interrupt handling disabled. RT-Linux implements preemptive priority and EDF scheduling mechanisms.

Turtle [5], designed in the High Performance Computing Laboratory at Mississippi State University, is based on RT-Linux and uses the RT-Linux's ability to intercept all hardware interrupts intended for the Linux kernel and to deliver them to the kernel only when the kernel is scheduled to run on the processor. Turtle uses the Intel Pentium Pro and following processors' integrated advanced programmable interrupt controller (APIC) [77] for fine-grained timing and interrupt control. Turtle provides a preemptive scheduling for periodic tasks using the Earliest Critical Deadline First (ECDF) algorithm. Under ECDF, each task periodic task specifies its period, maximum computation time, and relative deadline requirements. The scheduler prioritizes periodic task instances according to their deadlines, similar to the EDF algorithm. However, once a task instance has been allocated its maximum requested time and has not yet completed, the deadline of the next instance of the task is used in computing the task's priority (as opposed to the current instance's deadline). This allows other tasks with earlier deadline to preempt the task that has exceeded its computational requirements.

The KU real-time (KURT) [123] project modifications made to Linux enables explicit time-based control of Linux tasks. This is achieved primarily by increasing the frequency of Linux's timer. KURT's timing routines call the Linux timekeeping routines at the expected frequency in order to correctly maintain Linux time. Furthermore, in order to ensure that real-time tasks are invoked at the correct time, the scheduler's timer interrupt is programmed to occur 50 microseconds before the task's start time and the scheduler busy-waits until the actual start time occurs.

2.5 Scheduling in Non Real-Time Operating Systems

A *Beowulf* cluster [148] is a cluster constructed from commodity hardware (*i.e.*, personal computers) and popular operating systems such as Windows 2000 and Linux. However, these operating systems are designed for improving the responsiveness for interactive threads on individual nodes, and not for enhancing the performance of parallel applications.

2.5.1 Thread Scheduling in Windows 2000

Windows 2000 implements priority-driven preemption with round-robin scheduling at the thread level [144]. The highest priority ready thread is always given preference. An executing thread runs for a time interval determined by its current *quantum* value before being interrupted and replaced by another thread with the same priority. A thread can be preempted whenever a higher priority thread becomes executable. The quantum of a thread is an integer value initially set to 6 (12 on Windows 2000 Server version) when the thread is scheduled. At each timer interrupt, the quantum of the current thread is reduced by 3. If its quantum becomes equal to or less than zero, the current thread is preempted by another thread of equal priority. If no other threads of equal priority are executable, the current thread's quantum is reset and the thread resumes execution.

Whenever a thread enters a wait state (*i.e.*, calls *WaitForSingleObject* or *WaitForMultipleObjects*), its quantum is reduced by 1. This *quantum decay* process is utilized in order to ensure that threads that enter wait states have their quantum values reduced. Without quantum decay, a thread that repeatedly enters a wait state before the

timer interrupt occurs and resumes after the interrupt has been processed will never have its quantum reduced.

Windows 2000 boosts the quantum of all threads of the *foreground process* (*i.e.*, a process whose window has the input focus). This boost in quantum favors the foreground task while still giving the background tasks with the same priority a chance to execute.

The priority levels of threads can also be temporarily boosted after events such as I/O completion and if a thread has not been scheduled for a long time. Furthermore, threads can be interrupted at any time (although, the portions of interrupt handlers will not proceed until the current thread's *interrupt request level* (IRQL) is reduced to below the interrupt priority [125]). Windows 2000 does not provide real-time thread scheduling guarantees such as guaranteed interrupt latencies or guaranteed execution times. Threads with "real-time" Windows 2000 priorities are in similar to threads with conventional priority levels with the exception of having their quantum reset after they are preempted.

The threads' quantum and priority adjustments can cause significant variances in the execution time intervals of threads. Furthermore, the decisions to make priority and quantum adjustments are made by the individual instance of the OS running at each node without regard to the impact of such decisions on threads of a parallel application executing on other nodes. Therefore, executing a single parallel application with several competing and interacting threads or executing several parallel applications simultaneously on a Beowulf can result in lack of coscheduling and the concomitant loss of performance caused by context thrashing and resource contention.

2.5.2 Thread Scheduling in Linux

Linux also provides priority-driven, preemptive, round-robin thread scheduling [21]. Instead of directly using thread's quantum values to make scheduling decisions, Linux divides time into *epochs*. At the beginning of an epoch, a quantum value is computed and assigned to all threads. The quantum value determines the maximum amount of time the thread can execute in the epoch. An epoch ends when all executable threads (blocked threads are not considered) have used up their time quanta; at this point, the next epoch is initiated. Each process is initially assigned a base time quantum of 20 clock ticks. If a thread consumes its entire quantum during an epoch, the thread is given a new base time quantum at the beginning of the next epoch. If a thread does not use its entire quantum (*i.e.*, the thread was blocked when the epoch expired), the remaining quantum is used to compute a boost to the thread's quantum for the next epoch. This favors I/O bound threads.

Similar to Windows 2000, Linux cannot guarantee interrupt latencies and execution runtimes. Threads with "real-time" priority in Linux simply have a higher base priority than the conventional threads and may not receive a priority boost depending on the real-time scheduling policy class (*e.g.*, `SCHED_RR`) selected for the thread's process.

When looking for a new thread to schedule, the Linux scheduler uses the sum of the ready threads' base quantum value and the number of clock ticks remaining in the quantum to compute priorities. Therefore, thread priorities in Linux are dynamic and can lead to context thrashing and resource contention parallel applications in Beowulf applications.

2.5.3 Process Scheduling in K42

K42 [81] is a kernel based on the Tornado OS [60]. Tornado is an operating system specifically designed to improve OS performance on large-scale shared memory multiprocessor systems. Its design objective is to optimize performance by reducing cache coherency overheads and reducing data sharing. In order to enable an OS request to be serviced by the same processor on which the request was issued (and thereby preserving locality), OS data structures are distributed across the processors. For example, the various processors can simultaneously handle page fault requests to different pages. The use of distributed structures also enables the implementation of localized locks. This reduces the need for global shared locks across processors because locks are typically localized within single processors.

K42 is an extension of Tornado that enhances system performance by performing portions of the scheduling process in user-level code. Certain other OS services (*e.g.*, file systems) are also performed at the user level. This reduces application-kernel interaction overheads.

The kernel-level scheduler is only responsible for scheduling processes belonging to various applications for a limited time quantum. Within the process, the application is responsible for scheduling its own threads, and can optimize thread scheduling according to its own needs. When the quantum of a process expires, K42 first attempts a *soft-preemption*. In a soft-preemption, the executing process is interrupted and given some time to perform housekeeping. Essentially, the interrupted process maintains its own machine state. If the executing process has lower priority than the ready process, or has

not yielded the processor within the allowed time, K42 performs *hard-preemption* in which the executing process's machine state is saved in kernel space.

2.6 Real-Time Communication

In parallel real-time systems, on-time execution of communication operations is required in order to ensure the timely execution of real-time parallel applications' component tasks. Therefore, considerable research has been devoted to investigating techniques for admission control with resource reservation and developing protocols for mitigating contention for shared communication resources between competing real-time and non real-time tasks.

2.6.1 Admission Control and Resource Reservation

Admission control is the process of controlling the number of simultaneous connections or real-time *channels* between processors in a cluster. A real-time channel is a connection-oriented reservation of resources along the path between cooperating processes. Without admission control, the volume of network traffic offered at any given time is not controlled by a deterministic policy and this can result in congestion and large variances in communication delays that are unacceptable for real-time systems.

Tenet [59] uses a two-pass approach to reservation establishment. A *reservation* message containing the desired end-to-end delay is sent along the path from the receiver to the sender. Each switch along the path reserves resources and adds its delay into a field in the reservation message. If the cumulative delay is greater than the desired end-to-end delay, the sender rejects the request and all reservations associated with the

reservation request are released. If the cumulative delay is less than the desired end-to-end delay, the sender sends a *relax* message back along the original path in reverse. The relax message contains the excess delay and each switch along the path relax the reservations, increasing the delay to the extent possible consistent with meeting end-to-end delay bounds.

Ferrari has further refined the approach he proposed for Tenet [58]. This method is based on a simplex unicast real-time *channel* model that consists of a sequence of one or more queuing *servers* (*e.g.*, the CPU—NIC—NIC—CPU channel path to connect two cluster nodes consists of four servers). The admission test of processing capacity, buffer capacity, and delay bounds is performed on all nodes along the path before the channel is established and communication can proceed. This admission control method is an extension of an earlier approach proposed by Ferrari and adopted by Tenet.

The *Resource ReSerVation Protocol* (RSVP) [166, 167] uses a simpler one-pass approach to network reservation establishment. When the reservation message is sent from the receiver to the sender, the switches along the path make admission control decisions locally and do not record their locally induced delays onto the message. Fine-tuning of reservations is not performed because a confirmation or relax message is not sent along the reverse path as in Tenet. An error message is send along the reverse path only if a switch denies the reservation request. In this simple approach, the end-points of a complex network topology cannot specify end-to-end delay requirements to the network because the network does not record the delays introduced at each switch.

Further research conducted in the High Performance Computing Laboratory at Mississippi State University by Zhenqian Cui [40] provides theoretical and experimental analysis of the performance of RSVP. This analysis shows that while RSVP provides good average delay characteristics, it is unable to provide a tight bound on delay jitter. This is because the high-level RSVP protocol is unable to effectively control the lower-level link layers. Furthermore, the analysis shows that the processing overhead introduced by RSVP channel scheduling is significantly higher that a TCP connection in a high-speed network. His alternative approach using a global admission control policy that tracks global resource availability and utilization reduces delays caused by conflicts within switches, reduces scheduling overheads, and provides guaranteed end-to-end delays as long as the end systems provide sufficient processing times to the packet scheduling algorithms.

Xuan *et al*. [159] present an algorithm that makes run-time admission control decisions based on the current utilization of bandwidth and the bandwidth requirements of the connection being established. The configuration phase executed at system startup determines the maximum *safe* level of bandwidth utilized at each node. Maximum delay constraints of all traffic along all paths are guaranteed to be satisfied as long as utilization is at or below the safe level. The configuration phase uses the network topology, sender-receiver pairs together with traffic and maximum delay constraints to derive maximum safe utilization levels. At runtime, the admission control algorithm simply verifies that bandwidth is available along the path of a new connection.

Estimating network capacity at any given time is key to successful admission control and to estimate the end-to-end communication delays in parallel real-time environments. Banerjee and Agrawala [15] present a technique for estimating available *network capacity* (*i.e.*, the amount of data that can be inserted into the network at any given time) of an end-to-end connection using measurements taken at the endpoints only.

2.6.2 Access Arbitration and Transmission Control

When two or more nodes in a multi-access network simultaneously begin transmitting data over a single shared communication medium, all overlapping transmissions are lost when their signals *collide*. After a collision occurs, the failed communication operations must be retried, and repeated collisions can arbitrarily delay their successful completion. Media access control (MAC) techniques for mitigating delays caused by collisions have taken the form of *access arbitration*, *transmission control*, or both [112]. Access arbitration in real-time parallel systems is the process of granting communicating nodes access to the media in such a manner that collisions are avoided. Transmission control is the regulation of the length of time any one node can continue to exclusively transmit messages once it has access to the medium and is required in order to afford other nodes equal opportunity to transmit their own real-time messages.

Ethernet (IEEE 802.3) [23] uses a 1-persistent Carrier Sense Multiple Access with Collision Detection (CSMA/CD) with Exponential Binary Backoff protocol that can introduce large delays under high-load conditions, making it unsuitable for real-time computing. In order to avoid collisions, Venkatramani and Chiueh [154] have proposed,

RETHER, a token-based protocol implemented over Ethernet. This protocol imposes a token-based access arbitration policy in software when real-time service is required at any node. In this protocol, a special token packet circulates among the nodes in the network and only the node currently possessing the token is allowed to transmit. Nodes that have made bandwidth reservation are given priority over others. An admission control policy is used to prevent new nodes from establishing real-time sessions if there is insufficient remaining network capacity.

In order to avoid the occurrence of high load conditions, Kweon, Shin, and Zheng [88] present a static traffic "smoothing" approach based on the leaky bucket algorithm [38] to enforce a fixed rate of packet arrivals at the Ethernet MAC layer of a network node. Only non real-time packets are subjected to delays; real-time packets are forwarded to the MAC as soon as they appear. Kweon, Shin, and Workman [89], extend their previous approach by allowing nodes with higher outgoing traffic volume to adaptively increase rates of transmission when other nodes have reduced outgoing traffic. Instead of absolute real-time guarantees, these approaches provide statistical real-time channels over Ethernet. A statistical real-time channel guarantees that the probability of late packet delivery is less than a given constant.

Chou and Shin have also proposed a token-based approach for multi-access bus networks [33] that uses a centralized node to for admission control and token allocation in order to provide statistical real-time communication channels. This protocol can be readily adapted for use in Ethernet environments.

Real-time communication channels have also been implemented on token-based MAC protocols such as Token Ring (IEEE 802.5) [76] and Fiber Distributed Data Interface (FDDI) [4]. The Token Ring protocol implements a priority policy that allows nodes with higher priority to transmit first. Furthermore, transmission control is provided through the use of a maximum token hold time that limits the amount of time a node can transmit packets once it has acquired the token. FDDI is similar to Token Ring but does not provide for priorities; all nodes receive the token and transmit packets in a round-robin fashion. Kamat and Zhao [82] provide a detailed analysis of the real-time performance of Token Ring and FDDI networks using RMS for assigning priorities to nodes. Strosnider and Marchok [150] analyze real-time performance of Token Ring networks using DS scheduling.

Asynchronous Transfer Mode (ATM) [85] networks are also a popular choice for implementing real-time channels because ATM *cells* (*i.e.*, small – 53 bytes long – fixed length packets) are easy to switch and do not block other cells for very long. For the sake of efficiency and simplicity, most ATM switches implement simple FIFO or static priority scheduling schemes. Li, Bettati, and Zhao [102] investigate delay characteristics in priority scheduled ATM networks with arbitrary topologies in the face of possible *cyclic dependencies* among connections. Cyclic dependency is the results when the traffic on one connection interacts with another, causing unbounded delay in cell delivery [39]. Hansson, Ermedahl, and Tindell [70] and Ermedahl, Hansson, Sjődin [49] provide a method for admission control in ATM networks, and Ng, Song, and Zhao [122] propose a method for estimating the worst case cell delay in an ATM switch.

2.7 Scheduling of Parallel Tasks

Tasks and data dependencies in a parallel non-real-time application are often represented as vertices and edges, respectively, in a directed acyclic graph (DAG) [63, 91, 90]. A DAG $G = \{V, E\}$ consists of a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices and a set $E = \{e_1, e_2, \ldots, e_k\}$ of k directed edges connecting the vertices. The vertices represent computational tasks in a parallel application and the edges represent communication and precedence relations between the tasks. The ordered pair $e_i = (v_{src}, v_{dest})$ indicates that the direction of edge e_i is from vertex v_{src} to v_{dest}, implying that v_{src} must be completed before v_{dest} can begin. In a DAG, vertices without incoming edges are called *entry* vertices and vertices without outgoing edges are called *exit* vertices.

Given a DAG specifying a parallel application, the fundamental problem is to construct a series of assignment of tasks to processors such that the total time to complete the application is minimized without violating precedence or timing constraints. However, constructing optimal schedules from DAGs is NP-hard in general [48, 153]. Therefore, the successful application of systematic search techniques such as dynamic programming towards optimal DAG scheduling is impractical because of time and space constraints. A few cases of polynomial complexity DAG scheduling algorithms have been reported (*e.g.*, [74] and [135]) under simplifying conditions. However, these algorithms are of limited practical value because they are restricted to simple classes of DAGs and typically ignore communication costs.

Because of the intractability of DAG scheduling in general, a number of heuristic algorithms based on the *list scheduling* (LS) paradigm have been proposed that strive to

construct schedules that minimize makespan in polynomial time [67, 75, 86, 87, 91-92, 134, 157, 160]. List scheduling consists of repeatedly constructing an ordered list, called the *ready list*, of *ready tasks* and assigning the first task in this ready list to any available processor that can complete the task soonest. Ready tasks are as-yet-unscheduled tasks whose precedence constraints have been satisfied. Completion of executing tasks generates additional ready tasks that are inserted into the ready list. This scheduling process repeats until all tasks are completed. A variety of heuristics are used to determine the sequence of tasks in the ready list and to select the processor assignment for the task at the head of the ready list.

Kwok and Ahmad provide a comprehensive survey and taxonomy of heuristic list scheduling algorithms for statically scheduling DAGs [91]. The following is a brief description of popular list scheduling techniques.

The Edge-zeroing (EZ) algorithm [134] strives to minimize communication costs by coalescing vertices into clusters. Essentially, all edges are examined in descending order of weight and the two clusters connected by the largest edge are merged (and the weights of all interconnecting edges are reduced to zero) as long as the merger will not increase the starting time of the last vertex in the cluster beyond its *top-level* (t-level). The t-level of a vertex is the length of the longest path to the vertex from an entry vertex and specifies the earliest possible time this vertex can be scheduled. If the merger increases the starting time of the last vertex in the cluster beyond its t-level, one of the original clusters is scheduled to execute on a different processor. Within a cluster,

vertices are ordered by decreasing *bottom-levels* (b-level). The b-level of a vertex is the longest path from the vertex to a terminal vertex.

The Linear Clustering (LC) algorithm [86] determines the *critical path* (CP) of the DAG and merges the vertices on the CP into a cluster. The CP of a DAG is the longest path in the DAG form an entry vertex to an exit vertex. Any edges originating from or terminating at a vertex in the cluster are eliminated from further examination. The process is repeated by constructing the CP from the remaining vertices and edges until all edges are eliminated. Each cluster is scheduled to execute on a distinct processor.

Dominant Sequence Clustering (DSC) [160] uses the t-level and b-level metrics of the vertices to dynamically determine the critical path of the partially scheduled DAG during the scheduling process. This algorithm also schedules nodes to start as early as possible. Furthermore, Yang and Gerasoulis [161] prove that schedule length of DSC is bounded as follows:

$$M_{DSC} \leq \left(1 + \frac{1}{g}\right) M_{opt}, \tag{2.7}$$

where M_{DSC} is the length of the schedule produced by the DSC algorithm, M_{OPT} is the optimal schedule length for the DAG, and g represents the *granularity* of the DAG. Granularity is derived from the computation-to-communication ratio of the vertices and edges of the DAG. For coarse-grained DAGS (*i.e.*, with $g \geq 1$) the makespan of the scheduled created by DSC is less than or equal to twice the makespan of the optimal schedule.

The Mobility Directed (MD) algorithm [157] is a complex scheduling algorithm that does not fix the starting times of scheduled vertices until all nodes have been scheduled. Furthermore, vertices are scheduled in increasing order of their *relative mobility*. The relative mobility of a vertex provides an indication of the amount by which the vertex can be shifted to start at a later time so as to meet precedence constraints and start before its b-level. The relative mobility of vertex v_i is given by:

$$relative\_mobility(v_i) = \frac{Curr\_CP\_Length - (b\_level(v_i) + t\_level(v_i))}{w(v_i)}, \qquad (2.8)$$

where *Curr_CP_Length* is the length of the critical path of the partial schedule.

The Dynamic Critical Path (DCP) algorithm [93] outperforms other list scheduling algorithms in terms of reducing schedule lengths and processors used for many DAGs. At every LS step, DCP computes the CP from the partial schedule, and selects the ready vertex on the CP to be scheduled first. If none of the ready vertices is on the dynamic CP, the vertex with the least mobility is selected for scheduling. Mobility of a vertex is defined by:

$$mobility(v_i) = Curr\_CP\_Length - (b\_level(v_i) + t\_level(v_i)). \qquad (2.9)$$

DCP assigns vertex v_i to the process such that the sum of the start time of v_i and the start time of v_i's critical child is minimized over all processors that are scheduled to execute v_i's parent or child tasks. The critical child of v_i is that child vertex that maximizes the sum $w(v_i, v_c) + w(v_c)$ where v_c is in the set of child vertices of v_i.

The Highest Level First with Estimated Times (HLEFT) algorithm [3] orders the ready list in terms of the vertices' b-level computed once before scheduling begins. The

Insertion Scheduling Heuristic (ISH) algorithm [87] extends HLEFT by filling any "holes" left in the partial schedule after a vertex is scheduled by scheduling other vertices in the ready list (in order of priority) into the holes.

The Modified Critical Path (MCP) algorithm [157] uses the vertices' ALAP value to prioritize the ready list. Vertices with the smallest ALAP are scheduled first. A vertex is scheduled on a processor with the earliest available slot within which the node can be accommodated within precedence constraints.

The Earliest Time First (ETF) algorithm [75] uses a greedy heuristic that prioritizes vertices according to their earliest start times. The algorithm essentially examines every ready-vertex—processor pair exhaustively and schedules the ready vertex that can start the earliest on the processor that allows this earliest start time.

The Dynamic Level Scheduling (DLS) algorithm [139] recomputes the *dynamic levels* on all processors for every node in the ready list before each scheduling decision. The dynamic level (DL) of a processor-vertex pair is given by the difference between the b-level of the node and the earliest start time of the vertex on the processor. The processor-vertex pair with the largest DL is added to the final schedule.

A number of researchers have studied the efficacy of using genetic algorithms [72] for scheduling [19, 65, 66, 67, 107, 92, 116, 120, 133]. Genetic algorithms (GAs) search large, high dimensionality combinatorial spaces by emulating the evolutionary processes found in nature. Individuals are selected from a population of potential solutions to contribute their qualities to the next generation via a set of recombination operations (*e.g.*, *crossover* and *mutation*). Individuals with better characteristics as

determined by the optimization criteria are more likely to procreate. This process is repeated until a high-quality solution is found. The crossover operator determines how genetic (*i.e.*, schedule quality) information is exchanged between parents to construct better offspring. The mutation operator introduces randomness into the search that helps to escape from emerging local optima.

Kwok and Ahmad [92] have proposed an effective technique for combining a GA with LS. The combination of GA and LS is known as genetic list scheduling (GLS). In their GLS algorithm, a chromosome represents the order in which the tasks are scheduled. A schedule is constructed from a chromosome by scheduling each vertex in the order of appearance in the chromosome on the processor that allows the earliest start time. They also investigate the effectiveness of several crossover operators, the mutation operator, and adaptive genetic algorithm control parameters (*e.g.*, crossover rate, mutation rate, initial population size, and number of generations). Their parallel implementation outperforms the traditional heuristic algorithms described above in terms of both makespan and schedule construction time.

Grajcar [67] has presented a GLS algorithm for scheduling computation and communication in a heterogeneous multiprocessor system with shared communication busses. This algorithm also outperforms many of the traditional approaches described in literature. Grajcar has also described a physical heterogeneous hardware structure and a DAG structure for which most list schedulers produce poor-quality schedules [65].

2.8 Limitations of Existing Scheduling Research

Scheduling research for probabilistic soft real-time systems has been restricted to the assumption of periodicity and preemption. However the overhead cost of preemption is typically ignored in these analytical techniques. The PDFs for tasks' computation time requirements model the variability in the processing needs of the tasks and the variability of runtime caused by modern hardware features. The overhead introduced by preemption is essentially rolled into the PDF for task computation requirements. However, the total overhead of preemption for a task depends on the number of times the task is preempted, and therefore cannot be simply rolled into the tasks' computation time PDF *a priori*.

Periodic real-time scheduling research has also typically ignored the precedence relationships that exist between the various tasks in a real-time application. Task priorities and phasing (*i.e.*, different release times relative to each other within similar periods) are assumed to ensure that tasks are executed in the proper order. Furthermore, related sequences of tasks with identical periods, and consequently with identical priorities and deadlines, are typically coalesced into threads that are preemptively scheduled according to their priorities.

Another limitation of most existing analytical approaches to distributed soft real-time scheduling is the assumption that distributed tasks are organized as chains of subtasks that have been pre-allocated to processors. These chains of tasks are assumed not to split or merge. While a number of real-time communication techniques have been proposed that provide statistical guarantees on packet delivery latencies, there are two

fundamental shortcomings of this approach that limit its applicability to scheduling parallel and distributed applications.

The first shortcoming is that in complex parallel and distributed real-time applications, tasks often provide inputs to and receive inputs from multiple tasks. This implies that task chains split and merge in complex patterns. The second shortcoming is that the task clustering and processor allocation is left to the system designer. This can lead to an *ad hoc* design based on the system designers experience and bias that could require significant revision when system software, hardware, or constraints are modified.

The LS, GA, and GLS algorithms that have been developed for clustering and scheduling tasks in DAGs representing parallel applications provide a basis for the scheduling algorithms used in this dissertation. However, the existing research is limited to deterministic scheduling using WCET assumptions and has focused on reducing schedule lengths. Therefore, these previously developed algorithms cannot directly be employed in scheduling real-time systems where the schedule length minimization objective is secondary to meeting all deadlines.

Another LS, GA, GLS assumption that prevent their direct application to scheduling real-time systems is that the processors are assumed to be fully connected via point-to-point links. Therefore, it is assumed that multiple outgoing or multiple incoming communication operations over single processor-to-network links can occur without impacting each other. This is clearly not a realistic assumption unless a non-work-conserving time-division multiplexing scheme is used for communication. Most modern networks provide a single full-duplex link connecting the processor to the network. This

implies that communication operations corresponding to incoming/outgoing edges to/from vertices must be serialized, and the impact of this serialization must be considered during schedule construction.

The existing LS, GA, and GLS algorithms are also deterministic in nature and suitable for scheduling hard real-time systems where only WCET are assumed. There is no provision for using these techniques in situations where tasks have variable runtime requirements.

This dissertation presents scheduling techniques that over come the limitations of existing scheduling research in the following ways (these are explained in more detail in Chapter 3):

- New LS and GLS algorithms are proposed that strive to schedule tasks with non-deterministic runtime assumptions in order to produce schedules that trade off the probability of tasks completing within deadlines and the completion time jitter for reduced schedule lengths.
- Non-preemptive scheduling techniques are used to minimize the cost in terms of reduced performance and increased uncertainty cause by preemption.
- The scheduling algorithms strive to automatically organize the various parallel tasks into chains of tasks and to allocate these task chains to processors so as to produce optimal schedules.
- The scheduling algorithms are able to operate on applications that are represented in the form of DAGs with complex task chain patterns and precedence constraints.

- The scheduling algorithms assume a more realistic packet switched communication infrastructure as opposed to the fully connected point-to-point network assumed in most LS and GLS approaches.

CHAPTER III

STOCHASTIC TASK SCHEDULING APPROACH

This chapter describes the approach used to construct the stochastic schedules in this dissertation. The scheduling problem, modeled as a Directed Acyclic Graph (DAG) representation of real-time applications and a specification of the parallel platform model, is presented here. A theoretical framework for manipulating independent probability distribution functions (PDFs) in the context of scheduling is also developed. This theoretical framework is then used to describe how schedules that account for variable task execution times are constructed using a variety of novel heuristics for list scheduling and genetic list scheduling techniques. Other contributions include techniques for stochastic jitter control and techniques for systematically trading off probability of meeting end-to-end deadlines with schedule length and task completion time jitter.

3.1 Aperiodic Application Model

In traditional, deterministic non-preemptive scheduling of tasks with precedence constraints, the sets of tasks to be scheduled are often represented in the form of DAGs [91]. Figure 3.1 illustrates a simple hypothetical deterministic DAG. The amount of work to be performed in task v_i is represented as node weight w_{vi}. For example, in Figure 3.1, $w_{v0} = 2$. In homogeneous systems, the choice of processor used to execute a task does not affect the execution time of that task, and therefore, w_{vi} is used to represent the

execution time of the task. Similarly, edge weight w_{ei} represents the total work to be performed for edge e_i. In a homogeneous network environment, the choice of the source and destination processors does not affect the time taken to complete the communication operation. Therefore, w_{ei} is used to represent the communication time required by edge e_i.

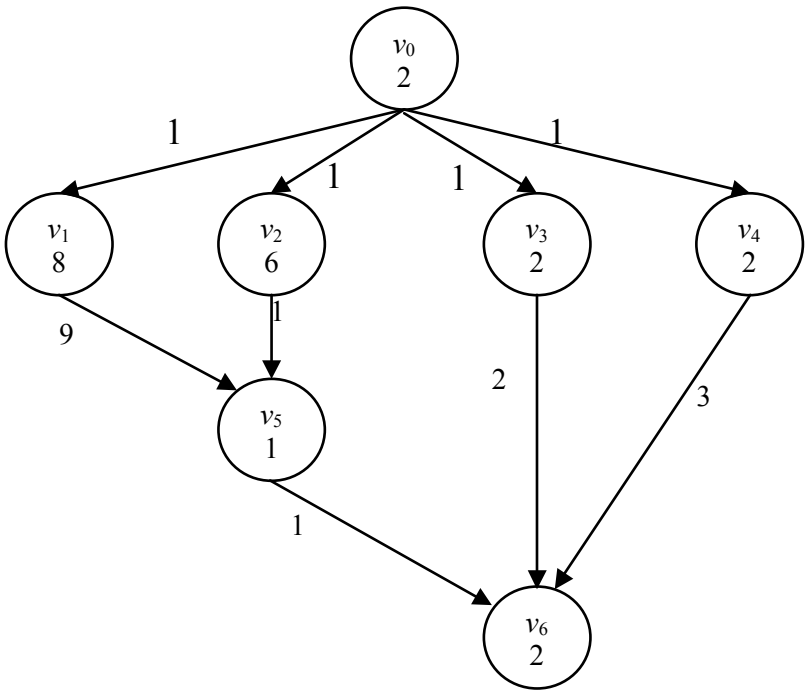
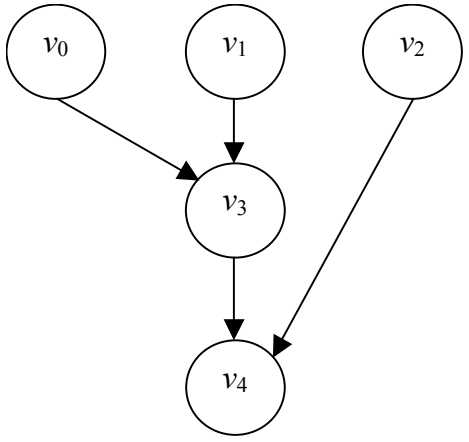


Figure 3.1 A Hypothetical DAG with Deterministic Task Weights

In this dissertation, if the source and destination processors are the same, the communication time is considered to be negligibly small (*i.e.*, w_{ei} is reduced to 0 when $p_{src}(e_i) = p_{dest}(e_i)$, where $p_{src}(e_i)$ and $p_{dest}(e_i)$ are the source and destination processors of the edge, respectively). The time taken to transfer data between tasks assigned to the same processor can be reduced significantly through the use of techniques such as

passing pointers to buffers between tasks as opposed to copying buffer contents. The Real-Time Message Passing Interface Standard [140] document provides specification for such advanced buffer management functionality.

DAGs can also be used to represent tasks with precedence constraints and varying execution requirements, as illustrated in Figure 3.2. The weight of the vertices and edges are assumed to be discrete, non-negative integer-valued, independent random variables.



Task	Weight Distribution					
v_0	Weight:	4	5	6		
	Probability:	⅓	⅓	⅓		
v_1	Weight:	7	8	9		
	Probability:	⅓	⅓	⅓		
v_2	Weight:	1	2	3		
	Probability:	⅓	⅓	⅓		
v_3	Weight:	2	3	4		
	Probability:	⅓	⅓	⅓		
v_4	Weight:	2	3	4		
	Probability:	⅓	⅓	⅓		
(v_0, v_3)	Weight:	1	2	3		
	Probability:	⅓	⅓	⅓		
(v_1, v_3)	Weight:	7	8	9		
	Probability:	⅓	⅓	⅓		
(v_2, v_4)	Weight:	1	2	3	4	5
	Probability:	0.5	0.35	0.05	0.05	0.05
(v_3, v_4)	Weight:	7	8	9		
	Probability:	⅓	⅓	⅓		

Figure 3.2 A Hypothetical DAG with Randomly Distributed Task Weights

The probability that a random variable X has value x, where $x \in \Im^+$ is represented by the notation $P(X = x)$. In general probability theory, random variables can take any value. However, in the context of this dissertation, execution time requirements, starting times, and completion times of real-time tasks can be restricted to be non-negative integers without loss of generality because of the following three reasons:

1. tasks cannot have negative execution time requirements,

2. all system events (*e.g.*, task start and completion) occur after the system start epoch that has a time value of 0 units, and
3. time measurements can be scaled to a finer resolution in order to result in integer values (*e.g.*, 1.5 seconds can be represented as 1500 milliseconds).

In DAGs with randomly distributed weights, every possible weight value for each task has an associated probability with which that weight occurs. In DAGs with deterministic weights, there is a single weight value for each task that occurs with 100% probability. Therefore, DAGs with deterministic weights are essentially special cases of DAGs with randomly distributed task weights.

Definition 1: The *probability distribution function* (PDF), also denoted as, $\pi_X(x)$, collectively specifies the probability that variable X will have the value x and can be viewed as an array of real numbers indexed by all possible values x of the random variable X. The real-valued array element specifies the probability of observing the array element's index value in an experiment. In order to meet the standard definition of probability distribution functions, $\forall x$: $\pi_X(x) \geq 0$ (*i.e.*, $\pi_X(x)$ must be non-negative for all values of x) and $\sum_x \pi(x) = 1.0$ (*i.e.*, the sum of $\pi_X(x)$ must be 1.0 over all values of x) [45].

In real-time systems, the domain of the π function is a time value (*e.g.*, the time at which a task begins, the time at which a task finishes, or the time required to complete a task). Furthermore, because real-time tasks are designed to complete in a finite amount of time, the PDF need only be defined over an interval $[l_X, u_X]$ for x. By definition, the range of the π function is a real number in the interval [0.0, 1.0]. Formally,

$$\pi_X(x) = \begin{cases} P(X = x) \text{ if } l_X \le x \le u_X \\ 0 \text{ otherwise.} \end{cases} \tag{3.1}$$

In this dissertation, the following notation is also used to denote a PDF:

$$\pi_X = \langle (x_1, r_1), (x_2, r_2), \cdots, (x_n, r_n) \rangle \tag{3.2}$$

where $x_i \in \Im^+$ and $r_i \in \Re^+$, and $1 \le i \le n$. Using this notation, the pair (x_i, r_i) is used to explicitly state that the probability that the random variable X will have a value of x_i is given by r_i.

Figures 3.3 and 3.4 depict example PDFs of the computational requirements of a simple matrix multiplication code using three nested loops. The PDFs were computed by measuring the number of CPU clock cycles required to complete the multiplication. The code was executed on a 450MHz Pentium III processor with disabled interrupts and no DMA operations. The matrices were all resident in memory (*i.e.*, virtual memory was disabled). However, paging and caching were enabled. Therefore, the variances in completion times are a result of processor features such as cache and TLB misses, and branch prediction failures. Interrupt handling and DMA cycle stealing did not play a role in determining the variances in completion times.

In Figure 3.3, a total of 65,536 runs were used to construct a histogram of the number of clock cycles required to complete the multiplication. The range of cycles required is [181,378,392, 182,030,457]. The probability for a task requiring a given number of cycles, t, is computed by dividing the number of times the program required t cycles to complete by 65,536 (the total number of samples).

In Figure 3.4, the PDF was scaled to a coarser-grain using microseconds as the unit of measurement, as opposed to clock cycles. This PDF was constructed by grouping completion times into 450-cycle "bins" (because for a 450MHz processor, an elapsed time of one microsecond is equivalent to 450 clock cycles). The range of microseconds required is [403,063, 403,163]. The probability for a task requiring a given number of microseconds, *t*, is computed by dividing the number of times the program required *t* microseconds by 65,535.

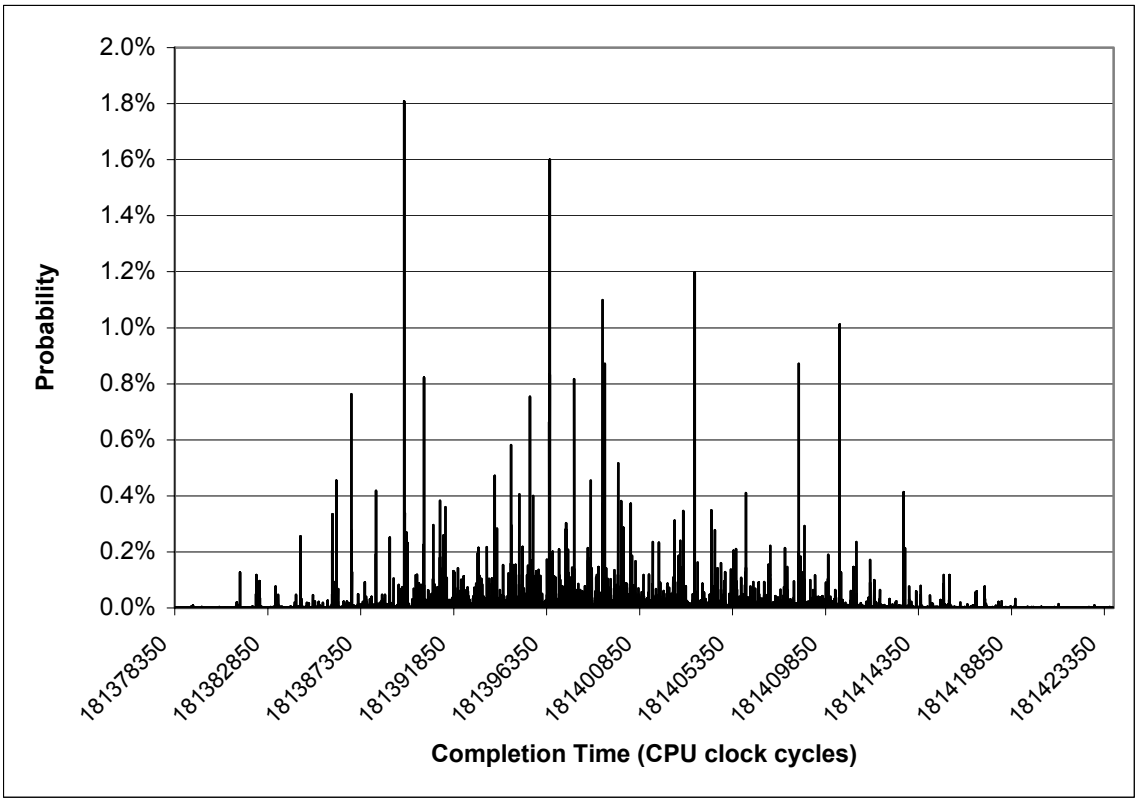


Figure 3.3 Example Fine-Grained PDF of an Integer Matrix Multiplication Task

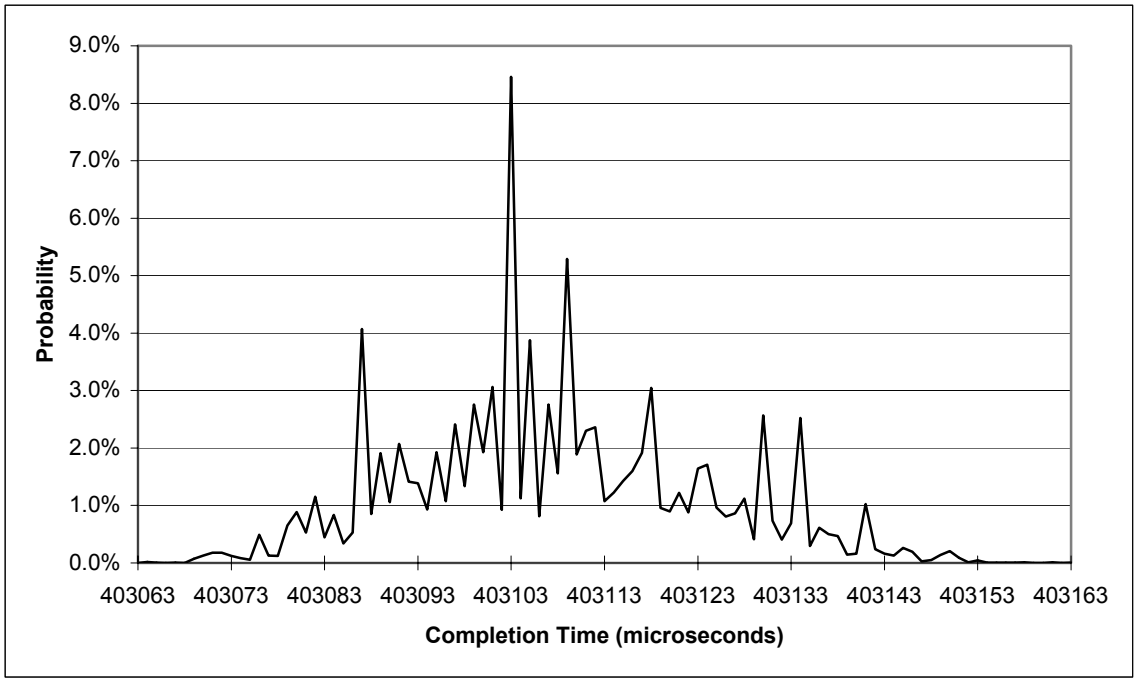


Figure 3.4 Example Coarse-Grained PDF of an Integer Matrix Multiplication Task

Definition 2: The *cumulative distribution function* (CDF) of random variable X is given by the following expression:

$$\Pi_X(x) = \sum_{l_X \le y \le x} \pi_X(y), \tag{3.3}$$

where l_X is the lower bound of the interval within which PDF π_X is defined. In other words, $\Pi_X(x)$ is the sum of all probabilities in the PDF array up to an including the probability at index value x. Furthermore, $\Pi_X(x) = P(X \le x)$. By definition, $\Pi_X(u_X) = 1$. Also, by definition, $P(X > x) = 1 - \Pi_X(x)$.

Definition 3: The *expected value* of a random variable X is given by the following expression [45]:

$$E[\pi_X(x)] = \sum_{l_X \le y \le u_X} y \cdot \pi_X(y), \tag{3.4}$$

where $[l_X, u_X]$ defines the interval over which PDF π_X is defined. The expected value represents the mean value of the random variable that will be obtained after running a number of experiments.

Definition 4: The *translation operation on a PDF* translates the domain of the PDF along the integer number line without affecting the range of the PDF. The $\oplus$ symbol is used to represent the translation operation. Formally,

$$\langle (i_1, r_1), (i_2, r_2), \ldots, (i_n, r_n) \rangle \oplus k = \langle (i_1 + k, r_1), (i_2 + k, r_2), \ldots, (i_n + k, r_n) \rangle, \tag{3.5}$$

where $k \in \Im$. The translation operation on a PDF also translates the expected value of the PDF by k units.

3.2 Periodic Application Model

The DAG model can also be used to represent periodic real-time applications comprised of a set of periodic tasks ϑ. The 4-tuple (w_i, ϕ_i, d_i, T_i) specifies the timing properties of periodic real-time task $J_i \in \vartheta$, where wi, ϕ_i, d_i, and T_i specify the computational requirements, phase, deadline, and period, respectively, of J_i. Given that periodic tasks imply an infinite sequence of repeated invocations, their scheduling is performed by analyzing task behavior within a *hyperperiod* (also called the *planning cycle*) [26]. Let L be the least common multiple (LCM) of the task periods (*i.e.*, λ_ϑ = LCM$\{T_i : J_i \in \vartheta\}$. For a set of periodic tasks with identical arrival times (*i.e.*, having identical ϕ_i), the length of the planning cycle is given by $\lambda_{pc}(\vartheta) = \lambda_\vartheta$. For a set of tasks

with arbitrary arrival times and $\phi_{max} = \max\{\phi_i : J_i \in \mathcal{J}\}$, the length of the planning cycle is given by $\lambda_{pc}(\mathcal{J}) = \phi_{max} + 2\lambda_{\mathcal{J}}$. Within the planning cycle, task J_i will be invoked $\lambda_{pc}(\mathcal{J})/T_i$ times. Individual invocations of the periodic tasks within the planning cycle can be viewed as distinct aperiodic tasks and the entire task set can be represented in the form of a DAG. The schedule constructed from the DAG is then executed repeatedly over the lifetime of the periodic application.

3.3 Parallel Platform Model

A parallel environment consisting of homogeneous processors and a homogeneous network is assumed for executing the parallel application. Therefore, the time taken to execute a computational task is the same on any processor. Also, a uniform network capacity is assumed over the entire parallel system. This implies that the time needed to complete a particular point-to-point communication operation is the same over any combination of source and destination processors.

Each processor is assumed to have a full duplex interface to the homogeneous virtual point-to-point network. A full duplex interface was selected for analysis in this dissertation because popular networking technologies such as Myrinet [120] and Ethernet [128] provide full duplex links. A full duplex interface allows a processor to simultaneously send and receive data over separate send and receive links. However, at each interface, the send and receive links operate in simplex mode. This restricts simultaneous communication to at most one incoming operation and one outgoing communication operation at each processor-network interface. While the PCI bus itself

is not full duplex [143], it is assumed that the fine-grained DMA bursts over the PCI bus combined with the available bandwidth for the DMA transfers results in minimal interference between simultaneously occurring send and receive communication operations.

Conversely, the switched network fabric is assumed to be contention free (*i.e.*, there are sufficient resources available to ensure that the various communication operations do not interfere with each other). This is a reasonable assumption because of the widespread availability of real-time communication services over a variety of networking technologies that can provide bounded latency on packet delivery [33, 58, 59, 70, 82, 88, 122, 150, 154, 159, 166, 167].

It is also assumed that computation and communication can be overlapped. In pragmatic systems, contention over shared system resources such as the system bus used by the network interface card (NIC) for both the outgoing and incoming DMA operations, and the shared system memory accessed by the CPU and the NIC can cause variances in completion times. However, these variances are assumed to be incorporated into the computation and communication tasks' PDFs.

3.4 Manipulating Probability Distribution Functions for Scheduling

During scheduling, the starting time of tasks depends on the completion time of any preceding tasks and the completion time of tasks depends on the starting time and computational requirements of the task. This section describes a variety of operations on PDFs that are useful for performing stochastic scheduling. The operations described in theorems 1-5, and lemmas 3 and 4 are new contributions of this dissertation.

Lemma 1: *Let $s_i(t)$ be the PDF of the starting time of task J_i and let $[l_s, u_s]$ be the interval over which PDF $s_i(t)$ is defined. Let $w_i(\tau)$ be the PDF of the execution time requirements of task J_i and let $[l_{wi}, u_{wi}]$ be the interval over which PDF $w_i(\tau)$ is defined. The time interval within which J_i will complete is given by $[l_f, u_f]$ such that*

$$l_{fi} = l_{si} + l_{wi} - 1, \text{ and} \tag{3.6}$$

$$u_{fi} = u_{si} + u_{wi} - 1. \tag{3.7}$$

Proof: Clearly, the earliest completion time of J_i occurs when it starts as early as possible and requires the least possible time to complete, resulting in equation (3.6). Similarly, the latest completion time of J_i occurs when it starts as late as possible and requires the maximum possible time to complete, resulting in equation (3.7). □

Corollary: *Let $f_i(t)$ be the given PDF of the completion times of task J_i and let $[l_f, u_f]$ be the interval over which $f_i(t)$ is defined. Also, let $w_i(\tau)$ be the PDF of the computational requirements of task J_i and let $[l_{wi}, u_{wi}]$ be the interval over which $w_i(\tau)$ is defined. The interval over which the PDF $s_i(x)$ of the starting times for J_i is defined is given by $[l_{si}, u_{si}]$ such that*

$$l_{si} = l_{fi} - l_{wi} + 1, \text{ and} \tag{3.8}$$

$$u_{si} = u_{fi} - u_{wi} + 1. \tag{3.9}$$

Proof: From equation (3.6), the earliest completion time of J_i is given by $l_{fi} = l_{si} + l_{wi} - 1$. Rearranging terms results in $l_{si} = l_{fi} - l_{ci} + 1$, which is equation (3.8). Similarly, equation (3.7) gives the latest completion time of J_i as $u_{fi} = u_{si} + u_{wi} - 1$. Rearranging terms results in $u_{si} = u_{fi} - u_{wi} + 1$, which is equation (3.9). □

Lemma 2: *Let $s_i(t)$ be the PDF of the starting time of task J_i and let $w_i(\tau)$ be the PDF of the computational requirements of task J_i. The PDF of the completion time of J_i can be computed by the convolution of $s_i(t)$ and $w_i(\tau)$ as follows* [18, 84]:

$$f_i(X) = \sum_{t=l_s}^{u_s} s_i(t) w_i(X - t + 1) \ . \tag{3.10}$$

Proof: From fundamental probability theory, task J_i will complete at time X when J_i begins at time t and when J_i requires exactly $X - t + 1$ time units to complete. Because t can be in the range $[l_{si}, u_{si}]$, there are $u_{si} - l_{si} + 1$ different combinations of start time and computation requirement times that can result in a completion time of X. The probability that a particular combination occurs of start time T and computation time $(X - T + 1)$ to result in a completion time of X is the product of the probability that J_i starts at time T and the probability that J_i requires $(X - T + 1)$ time to complete. Therefore, the overall probability that J_i completes at time X is given by the sum of probabilities of the individual combinations of start time and computation time requirements that result in a completion time of X. □

In this dissertation, the PDF convolution operator is denoted by the $\otimes$ symbol (*i.e.*, $f_i(x) = s_i(t) \otimes w_i(\tau)$).

The upper bound for the summation in equation (3.10) can be further refined based on the observation that when $t > X - l_w + 1$ (*i.e.*, $t = X - l_w + \tau + 1$ for all $\tau \geq 1$), the corresponding summation terms from equation (3.10) are given by the following:

$$\begin{aligned} s_i(t)w_i(X-t+1) &= s_i(X-l_w+\tau+1)w_i[X-(X-l_w+\tau+1)+1] \\ &= s_i(X-l_w+\tau-1)w_i(X-X+l_w-\tau-1+1) \\ &= s_i(X-l_w+\tau-1)w_i(l_w-\tau) \\ &= 0 \end{aligned} \tag{3.11}$$

The term $w_i(l_w - \tau)$ in equation (3.11) evaluates to a zero because $l_w - \tau$ is less than l_w, the lower bound of the weight PDF, for all $\tau \geq 1$. Therefore, the sum of products in equation (3.10) for values of $t > X - l_w + 1$ is zero. This implies that equation (3.10) can be rewritten as the following:

$$f_i(X) = \sum_{t=l_s}^{X-l_w+1} s_i(t)w_i(X-t+1) \ . \tag{3.12}$$

Theorem 1: *Given w_i, the PDF of completion times and the PDF of execution times of task J_i, the PDF of the starting times of J_i can be recursively computed as follows:*

$$s_i(T) = \frac{f_i(T+l_{wi}-1) - \sum_{t=l_{si}}^{T-1} s_i(t)w_i(T+l_{wi}-t)}{w_i(l_{wi})}; \ l_s \leq T \leq u_{si}. \tag{3.13}$$

Proof: The proof is by induction. For clarity, the subscript i identifying task J_i has been omitted in the following proof.

Basis: let $T = l_s$. By substitution, equation (3.13) gives the following:

$$\begin{aligned} s(l_s) &= \frac{f(l_s+l_w-1) - \sum_{t=l_s}^{l_s-1} s(t)w(l_s+l_w-t)}{w(l_w)} \\ &= \frac{f(l_s+l_w-1)}{w(l_w)}. \end{aligned} \tag{3.14}$$

Expanding $f_i(l_s + l_c)$ from (3.14) using equation (3.12) results in the following:

$$
\begin{aligned}
f(l_s + l_w - 1) &= \sum_{t=l_s}^{(l_s+l_w-1)-l_w+1} s(t)w(l_w + l_w - t) \\
&= \sum_{t=l_s}^{l_s} s(t)w(l_s + l_w - t) \\
&= s(l_s)w(l_s + l_w - l_s) \\
&= s(l_s)w(l_w).
\end{aligned} \tag{3.15}
$$

Substituting $s(l_s)w(l_w)$ for $f(l_s + l_w)$ into equation (3.14) results in the following equation proving the validity of the basis:

$$
\begin{aligned}
s(l_s) &= \frac{f(l_s + l_w)}{w(l_w)} \\
&= \frac{s(l_s)w(l_w)}{w(l_w)} \\
&= s(l_s)
\end{aligned} \tag{3.16}
$$

Inductive step: Assume equation 3.2 is true for n such that $l_s \le n < u_s$. Therefore,

$$
s(n) = \frac{f(n + l_w - 1) - \sum_{t=l_s}^{n-1} s(t)w(n + l_w - t)}{w(l_w)}; \; l_s \le n < l_u.
$$

The proof by mathematical induction requires that the following equation be true.

$$
\begin{aligned}
s(n+1) &= \frac{f[(n+1) + l_w - 1] - \sum_{t=l_s}^{(n+1)-1} s(t)w[(n+1) + l_w - t]}{w(l_w)} \\
&= \frac{f(n + l_w) - \sum_{t=l_s}^{n} s(t)w(n + 1 + l_w - t)}{w(l_w)}.
\end{aligned} \tag{3.17}
$$

Expanding $f(n + l_w)$ from equation (3.17) above gives the following equation:

$$
\begin{aligned}
f(n+l_w) &= \sum_{t=l_s}^{(n+l_w)-l_w+1} s(t)w(n+l_w-t+1) \\
&= \sum_{t=l_s}^{n+1} s(t)w(n+l_w-t+1) \\
&= s(n+1)w[n+l_w-(n+1)+1] + \sum_{t=l_s}^{n} s(t)w(n+l_w-t+1) \\
&= s(n+1)w(l_w) + \sum_{t=l_s}^{n} s(t)w(n+l_w-t+1)
\end{aligned}
\tag{3.18}
$$

Substituting the result of equation (3.18) into equation (3.17) results in the following equation:

$$
\begin{aligned}
s(n+1) &= \frac{f(n+l_w) - \sum_{t=l_s}^{n} s(t)w(n+1+l_w-t)}{w(l_w)} \\
&= \frac{s(n+1)w(l_w) + \sum_{t=l_s}^{n} s(t)w(n+l_w-t+1) - \sum_{t=l_s}^{n} s(t)w(n+1+l_w-t)}{w(l_w)} \\
&= \frac{s(n+1)w(l_w)}{w(l_w)} \\
&= s(n+1)
\end{aligned}
$$

Therefore, equation (3.17) is valid, which in turn proves the validity of equation (3.13). □

Definition 5: *Immediate predecessor tasks*: The immediate predecessor tasks of a task J_i in a DAG are those tasks that are directly connected to J_i and are followed by J_i. If task J_i is a vertex, then its immediate predecessors are the edges incident on J_i. For example, in Figure 3.1, the immediate predecessors of the computational task represented by vertex v_6 are the communication tasks represented by the edges (v_5, v_6), (v_3, v_6), and (v_4, v_6). If task J_x is an edge, then its only immediate predecessor task is the vertex at which the edge originates. For example, in Figure 3.1, the immediate predecessor of the

communication task represented by edge (v_0, v_1) is the computation task represented by vertex v_0.

Lemma 3: *Let X_1 and X_2 be two independent random variables with respective PDFs of $\pi_{X1}(t)$ and $\pi_{X2}(t)$ and respective CDFs of $\Pi_{X1}(t)$ and $\Pi_{X2}(t)$. The PDF of the maximum of the two variables is computed from the following expression:*

$$\pi_{\max(X1,\,X2)}(x) = \pi_{X1}(x)\pi_{X2}(x) + \pi_{X1}(x)\Pi_{X2}(x-1) + \Pi_{X1}(x-1)\pi_{X2}(x), \tag{3.19}$$

where $x \in [\max(l_{X1}, l_{X2}), \max(u_{X1}, u_{X2})]$.

Proof: Let $X = \max(X_1, X_2)$. The random variable X cannot have a value less than the maximum of the lowest values possible for variables X_1 and X_2 because the larger of the two values will be selected and returned by the max operation. Furthermore, the largest value of X is the largest value possible of X_1 and X_2 because neither random variable can contribute a larger value. Therefore, the interval over which X is defined is given by $[\max(l_{X1}, l_{X2}), \max(u_{X1}, u_{X2})]$.

Clearly, the value of X can be x if and only if one of the following three mutually exclusive events occurs:

1. both X_1 and X_2 simultaneously have a value of x, or
2. X_1 has a value of x and X_2 has a value less than x, or
3. X_1 has a value less than x and X_2 has a value of x.

Therefore, the probability that X has a value of x can be written as the following (noting that X_1 and X_2 are independent):

$$P(X = x) = P(X_1 = \mathrm{x})P(X_2 = \mathrm{x}) + P(X_1 = \mathrm{x})P(X_2 < \mathrm{x}) + P(X_1 < \mathrm{x})P(X_2 = \mathrm{x}). \tag{3.20}$$

Note that $P(X = x) \equiv \pi_{\max(X1, X2)}(T)$, $P(X_1 = \text{x}) \equiv \pi_{X1}(T)$, $P(X_2 = \text{x}) \equiv \pi_{X2}(T)$, $P(X_1 < \text{x}) \equiv \Pi_{X1}(x - 1)$, and $P(X_2 < \text{x}) \equiv \Pi_{X2}(x - 1)$. Making appropriate substitutions results in equation (3.19). □

Theorem 2: *Let X be the maximum of a set of n independent random variables* $\{X_1, X_2, \ldots, X_n\}$. *Let* $X_1, X_2, \ldots,$ *and* X_n *have PDFs* $\pi_{X1}(t), \pi_{X2}(t), \ldots,$ *and* $\pi_{Xn}(t)$, *respectively and CDFs* $\Pi_{X1}(t), \Pi_{X2}(t), \ldots,$ *and* $\Pi_{Xn}(t)$, *respectively. The PDF of the maximum of the n variables can be computed recursively as follows:*

$$\pi_{\max(X1, X2, \ldots, Xn)}(x) = \pi_{\max(X1, X2, \ldots, Xn-1)}(x)\pi_{Xn}(x) + \pi_{\max(X1, X2, \ldots, Xn-1)}(x)\Pi_{Xn}(x - 1) + \Pi_{\max(X1, X2, \ldots, Xn-1)}(x - 1)\,\pi_{Xn}(x). \quad (3.21)$$

Proof: The proof is by induction. Lemma 3 provides the basis for the inductive proof. Under the inductive step, equation (3.21) is assumed true, and the validity of the following expression must be demonstrated:

$$\pi_{\max(X1, X2, \ldots, Xn+1)}(x) = \pi_{\max(X1, X2, \ldots, Xn)}(x)\pi_{Xn+1}(x) + \pi_{\max(X1, X2, \ldots, Xn)}(x)\Pi_{Xn+1}(x - 1) + \Pi_{\max(X1, X2, \ldots, Xn)}(x - 1)\,\pi_{Xn+1}(x). \quad (3.22)$$

Let $Y = \max\{X_1, X_2, \ldots, X_n\}$. Y is also a random variable independent from X_{n+1}. Let $Z = \max\{Y, X_{n+1}\}$. From equation (3.19),

$$\pi_Z(x) = \pi_Y(x)\pi_{Xn+1}(x) + \pi_Y(x)\Pi_{Xn+1}(x - 1) + \Pi_Y(x - 1)\pi_{Xn+1}(x). \quad (3.23)$$

Substituting $\max\{X_1, X_2, X_n\}$ for Y in equation (3.23) results in equation (3.22). □

Theorem 2 suggests the iterative algorithm in Figure 3.5 for computing the PDF of the random variable resulting from taking the maximum of n independent random variables.

1. Let $X = \{X_1, X_2, \ldots, X_n\}$ be the set of n random independent variables.
2. Initialize $\pi_X := \pi_{X1}$.
3. loop $\forall X_i \in X - \{X_1\}$
4. $\quad l_X := \max(l_X, l_{Xi})$
5. $\quad u_X := \max(u_X, u_{Xi})$
6. $\quad$ loop $\forall x \in [l_X, u_X]$
7. $\quad\quad \pi_X(x) := \pi_X(x)\pi_{Xi}(x) + \pi_X(x)\Pi_{Xi}(x-1) + \Pi_X(x-1)\pi_i(x)$

Figure 3.5 Algorithm for Computing the Maximum PDF of a Set of PDFs

Note that because $\pi_{\max(X1, X2, \ldots, Xn)}(x)$ is essentially a sum of products and because multiplication and addition are commutative, the n random variables in the set can be processed in any order to produce the final PDF.

Definition 6: *Immediate successor tasks*: The immediate successor tasks of a task J_i in a DAG are those tasks that are directly connected to J_i and follow J_i. If task J_x is a vertex, then its immediate successors are the edges leading out of J_x. For example, in Figure 3.1, the immediate successors of the computational task represented by vertex v_0 are the communication tasks represented by the edges (v_0, v_1), (v_0, v_2), (v_0, v_3), and (v_0, v_4). If task J_x is an edge, then its only immediate successor task is the vertex at which the edge terminates. For example, in Figure 3.1, the immediate successor of the communication task represented by edge (v_0, v_1) is the computation task represented by vertex v_1.

Lemma 4: *Let X_1 and X_2 be two independent random variables with respective PDFs of $\pi_{X1}(t)$ and $\pi_{X2}(t)$ and respective CDFs of $\Pi_{X1}(t)$ and $\Pi_{X2}(t)$. The PDF of the minimum of the two variables is computed from the following expression:*

$$\pi_{\min(X1, X2)}(x) = \pi_{X1}(x)\pi_{X2}(x) + \pi_{X1}(x)[1 - \Pi_{X2}(x)] + [1 - \Pi_{X1}(x)]\pi_{X2}(x), \quad (3.24)$$

where $x \in [\min(l_{X1}, l_{X2}), \min(u_{X1}, u_{X2})]$.

Proof: Let $X = \min(X_1, X_2)$. The smallest value of X is the smallest value possible of X_1 and X_2 because neither random variable can contribute a smaller value. Furthermore, the random variable X cannot have a value greater than the minimum of the largest values possible for variables X_1 and X_2 because the smaller of the two values will be selected and returned by the min operation. Therefore, the interval over which X is defined is given by $[\min(l_{X1}, l_{X2}), \min(u_{X1}, u_{X2})]$.

Clearly, the value of X can be x if and only if one of the following three mutually exclusive events occurs:

1. both X_1 and X_2 simultaneously have a value of x, or
2. X_1 has a value of x and X_2 has a value greater than x, or
3. X_1 has a value greater than x and X_2 has a value of x.

Therefore, the probability that X has a value of x can be written as the following (noting that X_1 and X_2 are independent):

$$P(X = x) = P(X_1 = \mathrm{x})P(X_2 = \mathrm{x}) + P(X_1 = \mathrm{x})P(X_2 > \mathrm{x}) + P(X_1 > \mathrm{x})P(X_2 = \mathrm{x}). \qquad (3.25)$$

Note that $P(X = x) \equiv \pi_{\max(X1, X2)}(T)$, $P(X_1 = \mathrm{x}) \equiv \pi_{X1}(T)$, $P(X_2 = \mathrm{x}) \equiv \pi_{X2}(T)$, $P(X_1 > \mathrm{x}) \equiv 1 - P(X_1 \leq \mathrm{x}) \equiv 1 - \Pi_{X1}(x)$, and $P(X_2 < \mathrm{x}) \equiv 1 - P(X_2 < \mathrm{x}) \equiv 1 - \Pi_{X2}(x)$. Making appropriate substitutions results in equation (3.24). □

Theorem 3: *Let X be the minimum of a set of n independent random variables* $\{X_1, X_2, \ldots, X_n\}$. *Let* $X_1, X_2, \ldots,$ *and* X_n *have PDFs* $\pi_{X1}(t), \pi_{X2}(t), \ldots,$ *and* $\pi_{Xn}(t)$, *respectively and CDFs* $\Pi_{X1}(t), \Pi_{X2}(t), \ldots,$ *and* $\Pi_{Xn}(t)$, *respectively. The PDF of the minimum of the n variables can be computed recursively as follows:*

$$\pi_{\min(X1, X2, \ldots, Xn)}(x) = \pi_{\min(X1, X2, \ldots, Xn-1)}(x)\pi_{Xn}(x) + \pi_{\min(X1, X2, \ldots, Xn-1)}(x)[1 - \Pi_{Xn}(x)] + [1 - \Pi_{\min(X1, X2, \ldots, Xn-1)}(x)]\,\pi_{Xn}(x). \quad (3.26)$$

Proof: The proof is by induction. Lemma 4 provides the basis for the inductive proof. Under the inductive step, equation (3.26) is assumed true, and the validity of the following expression must be demonstrated:

$$\pi_{\min(X1, X2, \ldots, Xn+1)}(x) = \pi_{\min(X1, X2, \ldots, Xn)}(x)\pi_{Xn+1}(x) + \pi_{\min(X1, X2, \ldots, Xn)}(x)[1- \Pi_{Xn+1}(x)] + [1- \Pi_{\min(X1, X2, \ldots, Xn)}(x)]\,\pi_{Xn+1}(x). \quad (3.27)$$

Let $Y = \min\{X_1, X_2, \ldots, X_n\}$. Y is also a random variable independent from X_{n+1}. Let $Z = \min\{Y, X_{n+1}\}$. From equation (3.24),

$$\pi_Z(x) = \pi_Y(x)\pi_{Xn+1}(x) + \pi_Y(x)\Pi_{Xn+1}(x - 1) + \Pi_Y(x - 1)\pi_{Xn+1}(x). \quad (3.28)$$

Substituting $\max\{X_1, X_2, X_n\}$ for Y in equation (3.28) results in equation (3.27). □

Theorem 3 suggests the iterative algorithm in Figure 3.6 for computing the PDF of the random variable resulting from taking the minimum of n independent random variables. Note that because $\pi_{\min(X1, X2, \ldots, Xn)}(x)$ is essentially a sum of products and because multiplication and addition are commutative, the n random variables in the set can be processed in any order to produce the final PDF.

1. Let $X = \{X_1, X_2, \ldots, X_n\}$ be the set of n random independent variables.
2. Initialize $\pi_X := \pi_{X1}$.
3. loop $\forall X_i \in X - \{X_1\}$
4. $\quad l_X := \min(l_X, l_{Xi})$
5. $\quad u_X := \min(u_X, u_{Xi})$
6. $\quad$ loop $\forall x \in [l_X, u_X]$
7. $\quad\quad \pi_X(x) := \pi_X(x)\pi_{Xi}(x) + \pi_X(x)[1 - \Pi_{Xi}(x)] + [1 - \Pi_X(x)]\pi_i(x)$

Figure 3.6 Algorithm for Computing the minimum PDF of a Set of PDFs

Theorem 4: *Given the release time PDF, $r_i(t)$, and the corresponding CDF, $R_i(t)$, of task J_i, if the starting time of J_i is further restricted to be no earlier than some arbitrary time T, the starting time PDF of J_i is given by the following equation:*

$$s_i(t) = \begin{cases} 0 \text{ if } t < T, \\ R_i(t) \text{ if } t = T, \\ r_i(t) \text{ if } t > T. \end{cases} \tag{3.29}$$

Proof: The proof is by construction. There are three distinct cases as follows:

1. When $t < T$: because J_i's release is delayed until time T, any probability J_i has of being started before time T under PDF $r_i(t)$ is reduced to zero (*i.e.*, J_i cannot be started before T). Therefore, $s_i(t) = 0$ if $t < T$.
2. When $t = T$: if J_i were to be released before T under $r_i(t)$, then J_i will be delayed and started at T. Therefore, all probabilities under $r_i(t)$ that J_i is released before T are summed into the probability that J_i is started at T under $s_i(t)$. By definition, $P(J_i \text{ is released before } t \text{ under } r_i) + P(J_i \text{ is released at } t \text{ under } r_i) = R_i(t)$. Therefore, $s_i(t) = R_i(t)$ if $t < T$.
3. When $t > T$: if J_i is to be released after T under $r_i(t)$ then the restriction on start time has no effect, and J_i will be started as permitted by the release time PDF. Therefore, $s_i(t) = r_i(t)$ if $t > T$.

The three cases combined result in equation (3.29). □

Corollary: *Task J_i's start time jitter is reduced when its start time is restricted to occur no sooner than time $T > l_{ri}$, where $[l_{ri}, u_{ri}]$ is the interval of release times for J_i.*

Under this restriction, the interval for start times for J_i becomes $[l_{si}, u_{si}] = [max(l_{ri}, T), max(u_{ri}, T)]$. If $T < l_{ri}$, then according to theorem 2, there is no impact on the start time PDF (*i.e.*, $\pi_{si} \equiv \pi_{ri}$), and concomitantly, there is no change in the release time intervals. If $l_{ri} < T < u_{ri}$, the start time PDF's new interval becomes $[T, u_{ri}]$, and because $l_{ri} < T$, the new interval range is smaller than the original range, thereby reducing the release time jitter. If $T \geq u_{ri}$, the new start time interval becomes, $[T, T]$, reducing the jitter to 0.

Theorem 5: *Given the completion time PDF, $f_x(t)$, and starting time CDF, $S_y(\tau)$, of two successive tasks in a schedule, J_x and J_y, respectively. The probability that J_x will complete before J_y is scheduled to start is given by the following expression:*

$$\pi = \sum_{t=l_{fx}}^{u_{fx}} f_x(t)(1 - S_y(t)), \tag{3.30}$$

where $[l_{fx}, u_{fx}]$ is the interval over which f_x is defined.

Proof: let A and B be the random variables representing the completion time for J_x and the scheduled starting time for J_y, respectively. Therefore,

$$P(A < B) = \sum_{t=l_{fx}}^{u_{fx}} P(A = t)P(B > t).$$

By definition, $P(B > t) = (1 - P(B \leq t))$. Therefore,

$$P(A < B) = \sum_{t=l_{fx}}^{u_{fx}} P(A = t)(1 - P(B \leq t)). \tag{3.31}$$

Also by definition, $P(A = t) \equiv f_x(t)$ and $P(B \leq t) \equiv CDF(B) \equiv S_y(t)$. Making appropriate substitutions in equation (3.31) results in equation (3.30). □

3.5 Stochastic Scheduling Overview

In the context of this dissertation, a stochastic schedule is the temporal allocation of resources to tasks; CPUs to vertices and communication links to edges, in particular. In other words, each task in a DAG is mapped onto a particular resource and is "dispatched" for execution within at a predetermined range of time specified by the task's start time PDF. Furthermore, the task is expected to complete within a predetermined range of times specified by the task's completion time PDF. Note that tasks are started as soon as the preceding tasks have completed; a task's the start time PDF merely specifies the probability with which the task will be started at any given time. This section describes how the start time and completion time PDFs are computed in the stochastic DAG scheduling algorithms investigated in this dissertation. Sections 3.6 and 3.7 describe how the tasks' start and completion times influence the mapping of tasks onto resources using list scheduling and genetic list scheduling approaches, respectively.

3.5.1 Computing Schedule Start Times for Vertices

The start time PDF of a task in the DAG is computed from the completion time PDFs of the immediately preceding tasks using the algorithm in Figure 3.5 derived from Lemma 3 and Theorem 2. Essentially, the start time PDF of a vertex v_i on processor p is determined by the following expression:

$$s_{vi} = \max(\{\langle(0,1.0)\rangle\} \bigcup \{f_{vp}\} \bigcup \{f_e : e = (v_y, v_i) \in E \wedge p_{src}(e) \neq p\}) \oplus 1, \qquad (3.32)$$

where f_{vp} is the completion time for vertex, v_p, the *immediate schedule predecessor* of v_i scheduled on p, and f_e is the completion time PDF of an edge e with that is incident on

vertex v_i. A vertex v_p is an immediate schedule predecessor of v_i if v_i follows v_p in the schedule for processor p and no other vertex is scheduled on p between the completion of v_p and the starting of v_i. The PDF $\langle(0, 1.0)\rangle$ in equation (3.32) accounts for the case that a vertex with no preceding edges can be scheduled on a processor at time unit 1 if there are no vertices already scheduled to start at time unit 1 on that processor. Furthermore, only those edges incident on vertex v_i originating from processors other than the processor on which v_i is to be scheduled are included in the computation of the start time for v_i. This is because the execution time requirements for edges whose originating vertices are also scheduled on processor p are reduced to zero, making their finish time PDFs be the same as the finish time PDFs of the originating vertices. The finish time of the sequence of all preceding vertices of v_i that are scheduled on processor p are incorporated into f_{vp}.

For example, consider the DAG in Figure 3.2, and its schedule in Figure 3.4; vertex v_3 can only start after all three of the following occur:

1. The completion of any vertex v_k that may be already using the processor on which v_3 is scheduled at the time v_3 becomes ready. In this example, $v_k \equiv v_1$.
2. The completion of edge (v_0, v_3).
3. The completion of edge (v_1, v_3).

In this example, the weight of edge (v_1, v_3) is considered to be negligible because v_1 and v_3 are allocated to the same processor. Therefore, the completion PDF of (v_1, v_3) is identical to the completion PDF of v_1. It is important to note that Lemma 3 and Theorem 2 only apply to independent random variables. Therefore, the completion time PDF of v_1

can only be used once in equation (3.32). This implies that the starting time PDF of v_3 is as follows:

$$s_{v3} = \max\{f_{v1}, f_{(v0,\,v3)}\} \oplus 1, \tag{3.33}$$

and the completion time of v_3 is computed using Lemma 2 as follows:

$$f_{v3} = s_{v3} \otimes w_{v3}. \tag{3.34}$$

3.5.2 Computing Schedule Start Times for Edges

Consider edge (v_i, v_j) to be scheduled between processors p_{src} and p_{dest} (*i.e.*, vertex v_i is scheduled on processor p_{src} and vertex v_j is to be scheduled on processor p_{dest}). The edge will occupy a time slot in the schedule for the send link on p_{src} and a time slot in the schedule for the receive link on p_{dest}. Furthermore, these time slots will have identical starting and completion time PDFs. Let e_a and e_b be the edges whose completion determines the starting point of the idle time slot in the send link on p_{src} and receive link on p_{dest} into which edge (v_i, v_j) is to be scheduled. The start time PDF of (v_i, v_j) is determined using the following expression:

$$s_{(vi,vj)} = \begin{cases} f_{vi} & \text{if } p_{src} = p_{dest}, \\ \max(\{f_{vi}\} \bigcup \{f_{ea}\}) \oplus 1 & \text{if } e_a = e_b, \text{and} \\ \max(\{f_{vi}\} \bigcup \{f_{ea}, f_{eb}\}) \oplus 1 & \text{otherwise} \end{cases} \tag{3.35}$$

where f_{vi} is the completion time for vertex, v_i, the immediate predecessor for (v_i, v_j) and f_{ea} and f_{eb} are the completion time PDFs of edges e_a and e_b. The first part of equation (3.35) accounts for the case when the edge does not need to be scheduled because vertices v_i and v_j are scheduled on the same processor. The second part of equation (3.35) accounts for the special case when e_a and e_b are the same edge (*i.e.*, a single edge determines the

starting point of the idle slot in the send and receive links of p_{src} and p_{dest}, respectively). In this case, the starting PDFs of the idle slots in the send and receive links are identical, and therefore, dependent. Consequently, the idle slot starting PDF is used only once in the *max* operation.

As an example, consider the DAG in Figure 3.2 and the corresponding schedule in Figure 3.4; edge (v_0, v_3) can only start after the following events occur:

1. The completion of vertex v_0, the source vertex of (v_0, v_3).
2. The completion of edge (v_2, v_4) because (v_0, v_3) and (v_2, v_4) are scheduled to be received on processor 1 over the same receive link and (v_2, v_4) is scheduled to occur before (v_0, v_3) can begin.

Note that because no other edges are scheduled on the send link on processor 0 (the source of (v_0, v_3)), the starting time of (v_0, v_3) is given by the following:

$$s_{(v0,\,v3)} = \max\{f_{v0}, f_{(v2,\,v3)}\} \oplus 1, \tag{3.36}$$

and the completion time of (v_0, v_3) is computed using Lemma 2 as follows:

$$f_{(v0,\,v3)} = s_{(v0,\,v3)} + w_{(v0,\,v3)}. \tag{3.37}$$

3.6 List Scheduling Approach

An important contribution of this dissertation is the generalization of deterministic list scheduling approaches for non-preemptive scheduling of soft real-time applications with variable task execution time requirements and inter-task precedence constraints. In this section, three different heuristics for stochastic list scheduling are developed.

The fundamental LS algorithm consists of the steps outlined in Figure 3.7. The key steps in this algorithm are the prioritization of the vertices in the ready list and the scheduling of vertices and associated preceding edges on the processors. Prioritization of vertices is performed according to a variety of heuristics (*e.g.*, prioritize vertices in non-increasing order of paths leading from the vertices to the terminal vertex, or prioritize vertices in order of their earliest start times on any available processor). The prioritization heuristic selected can have a profound impact on the length of the schedule and the total amount of time required in constructing the schedule (as is show in [90 - 93] and in the experimental results of this dissertation in Chapter IV).

1. Construct a *ready list* of vertices with no preceding vertices.
2. Loop while vertices remain in the ready list:
3. Prioritize the ready list.
4. Remove the highest priority vertex from the ready list and schedule it on the processor that will allow the earliest start time for this vertex.
5. Add the newly readied vertices to the ready list.

Figure 3.7 The Fundamental List Scheduling algorithm

The three novel vertex prioritization heuristics based on random task execution time requirements that were developed as part of this dissertation are discussed below. New algorithms for mapping resources to tasks based on stochastic task release times while accounting for contention over communication links are also describe in detail below.

3.6.1 Stochastic Highest Level First with Estimated Times

The *Stochastic Highest Level First with Estimated Times* (SHLEFT) heuristic is a direct adaptation of the HLEFT heuristic [3] used in deterministic scheduling. In the HLEFT heuristic, the fixed WCET of each vertex and edge is used to compute the b-levels for the vertices and the vertices in the ready list are scheduled in a non-decreasing order of their b-levels. In the SHLEFT approach, the expected values of the weight PDFs of all tasks in the DAG are used to compute the stochastic b-levels for the vertices. As in HLEFT, the vertices in the ready list are scheduled in a non-decreasing order of their stochastic b-levels.

3.6.2 Stochastic Earliest Time First

The *Stochastic Earliest Time First* (SETF) heuristic is based on the greedy ETF heuristic used in deterministic scheduling [75]. Under ETF, every ready vertex is tentatively scheduled on every available processor. The ready vertex that can be started the earliest on any available processor is selected for scheduling first. The SETF is identical to the ETF approach except that the expected values of the PDFs of the starting times of the ready vertices are used to prioritize the ready vertices (*i.e.*, the ready vertex with the earliest expected start time on any processor is selected for scheduling first).

3.6.3 Stochastic Critical Path

Definition 7: The list of vertices of a DAG is said to be in *topological order* if for all pairs of vertices (v_i, v_j) in the DAG such that v_i appears before v_j in the list, there is no path in the DAG leading from v_j to v_i [90].

The *Stochastic Critical Path* (SCP) heuristic is roughly based on the MD heuristic [157] used for deterministic scheduling. Under the SCP heuristic, the *stochastic mobility* of each vertex is computed from the weight PDFs of the tasks in the DAG and ready vertices are scheduled in the order of non-decreasing stochastic mobility. The algorithm for computing the stochastic mobility attribute of a vertex is given in Figure 3.1.

```
1.  Let V := list of vertices in the DAG
2.  Let critical_path_length := 0
3.  Loop ∀v ∈ V taken in topological order
4.      Let E_pv := list of immediate predecessor edges of v
5.      v.earliest_start_PDF := max({⟨(0, 1.0⟩)} ∪ {e.earliest_end_PDF : e ∈ E_pv}) ⊕ 1
6.      v.earliest_end_PDF := v.earliest_start_PDF ⊗ v.weight_PDF
7.      Let E_sv := list of immediate successor edges of v
8.      Loop ∀e′ ∈ E_sv
9.          e′.earliest_start_PDF := v.earliest_end_PDF ⊕ 1
10.         e′.earliest_end_PDF := e′.earliest_start_PDF ⊗ e′.weight_PDF
11.     if E_sv = ∅ ∧ v.earliest_end_pdf.expected_value > critical_path_length
12.             critical_path_length : = v.earliest_end_pdf.expected_value

13. Loop ∀v ∈ V taken in reverse topological order
14.     Let E_sv := list of immediate successor edges of v
15.     if E_sv = ∅
16.         v.latest_end_time := critical_path_length
17.     else
18.         v.latest_end_time := min{E[e′.earliest_end_PDF]: e′ ∈ E_sv} − 1
19.     v.latest_start_time := v.latest_end_time – E[v.weight_PDF] + 1
20.     Let E_pv := list of immediate predecessor edges of v
21.     Loop ∀e ∈ E_pv
22.         e.latest_end_time := v.latest_start_time – 1
23.         e.latest_start_time := e.latest_end_time – E[e.weight_PDF] + 1

24.     v.mobility := v.latest_start_time – E[v.earliest_start_PDF]
```

Figure 3.1 Algorithm for Computing the Stochastic Mobility Attribute of Vertices

The stochastic mobility of a vertex is essentially the difference between the expected value of the vertex's *earliest start time PDF* and the vertex's *latest start time* attribute. The earliest start time PDF of each vertex is computed in a forward pass through the DAG in topological order. The earliest start time PDF of a vertex is computed by taking the maximum of the *earliest end time PDFs* of the vertex's immediate predecessor edges and translating the result to the right (*i.e.*, increasing the PDF domain) by one time unit. The *earliest end time PDF* of a vertex is computed by convoluting the earliest start time PDF of the vertex by the vertex's weight PDF. The earliest start time PDF of an edge is computed by translating the earliest end time PDF of the source vertex of the edge by one unit to the right. The earliest end time PDF of the edge is computed by convoluting the earliest start time PDF of the edge with the edge's weight PDF. The length of a critical path in the DAG is given by the maximum of the expected values of the *terminal vertices* in the DAG. A terminal vertex is one that does not have any outgoing edges.

During the forward pass, the vertices and edges of the DAG are not scheduled on the processors, and the edges are not serialized (*i.e.*, edge execution times may be overlapped with each other and resource utilization restrictions are ignored). Therefore, the earliest starting time PDFs of vertices and edges are not suitable for use in scheduling and are used only indirectly as a heuristic to guide the scheduling process.

The latest end time attribute and the stochastic mobility attribute of vertices are computed in a backwards pass through the DAG in reverse topological order. The latest end time attribute of a terminal vertex is the same as the DAG's critical path length. The

latest end time attribute of a non-terminal vertex is computed by selecting the minimum of the *latest start time* attributes of the vertex's immediate predecessor edges and subtracting one from the result. Subtracting 1 from the latest start time attribute of the edge's immediate predecessor vertex results in the latest end time attribute of an edge. The latest start time attribute of edges and vertices are computed by subtracting the expected value of their weights from the corresponding latest end time attributes.

3.6.4 Resource Allocation

The process of scheduling a vertex requires first allocating all preceding communication tasks (edges) and then locating an idle time interval, in the schedule of the processor, beginning at or after the task's release time, that is of sufficient length to accommodate the task. Note that the preceding communication tasks are scheduled immediately after the candidate processor for the computation task is selected.

The process of scheduling an edge requires locating time intervals in the schedules of the communication links at the source and destination processors such that the intersection of the idle intervals is of sufficient length to accommodate the communication task. Any portion of the idle interval that lies before the release time of the communication task is not considered available for scheduling. This ensures that the communication task is not scheduled to begin before its release time. Because communication between two computation tasks scheduled on the same processor consumes no time and network resources, no interval lookup is required (*i.e.*, the communication is assumed to occur instantaneously).

For reasons of simplicity, resource allocation using deterministic task weights is discussed first. Figure 3.2 illustrates the Gantt chart for the optimal deterministic schedule constructed for the DAG in Figure 3.1. Note that communication tasks occupy *matching* intervals in the corresponding link schedules (*i.e.*, the intervals have the same start time and duration).

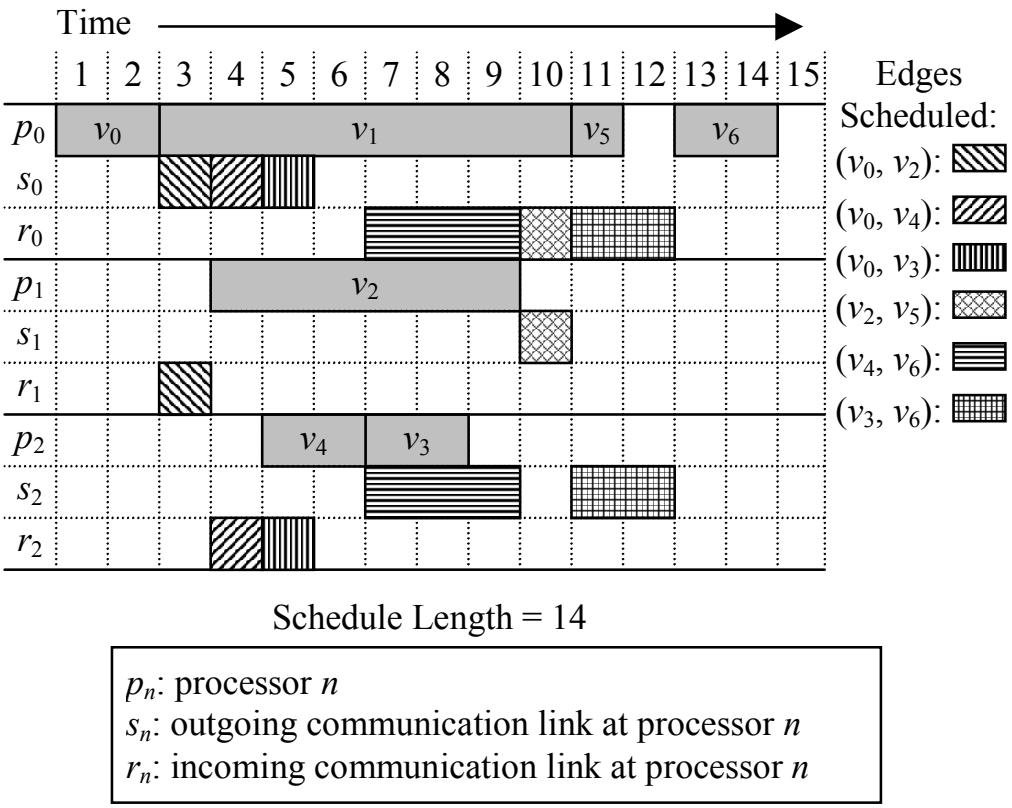


Figure 3.2 Gantt Chart for the Optimal Schedule for the DAG in Figure 3.1

The key to efficient scheduling with LS is to prioritize the vertices in the ready list correctly and to prioritize the order in which the preceding edges are scheduled. For example, Figure 3.3 illustrates the non-optimal schedule for the DAG in Figure 3.1 when

vertex v_3 is scheduled before vertex v_4. The schedule is longer in Figure 3.3 because edge (v_4, v_6) must wait until time unit 11 to begin (after the previously scheduled edge (v_2, v_5) completes). Note that although the sending communication link is available to (v_4, v_6) starting at time unit 9, (v_4, v_6) must wait until time unit 11 because the receiving communication link is busy with (v_2, v_5) at time unit 10.

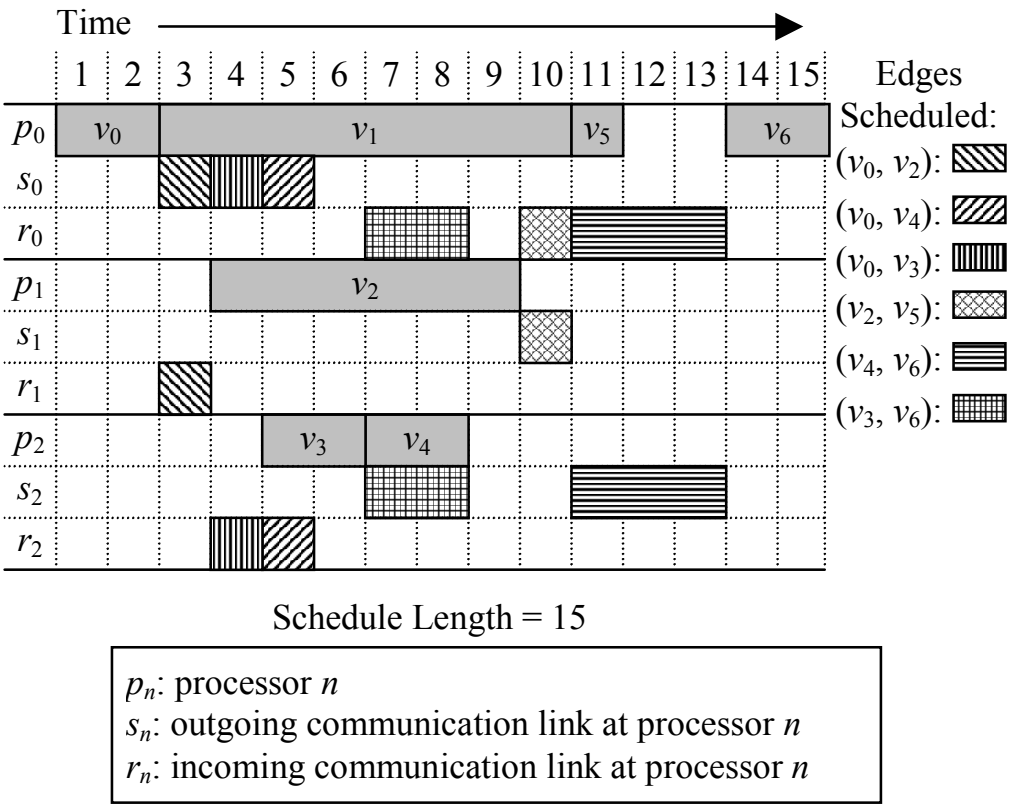


Figure 3.3 A Non-Optimal Schedule when v_3 is Scheduled before v_4

In the LS approaches used this dissertation, edges are always scheduled in order of non-decreasing weight. While this "first fit decreasing" heuristic does not guarantee optimality, it has been shown to perform well in general for "bin packing" problems [12].

In the genetic list scheduling (GLS) approach discussed in Section 3.7, the GA procedure dynamically determines edge-scheduling priorities, as opposed to using a static non-decreasing order.

The steps described above in constructing deterministic schedules from tasks with fixed execution time requirements can be generalized to solve the problem of stochastic schedule construction for tasks with varying execution time requirements. It is important to note that constructing schedules for tasks with fixed execution time requirements is a special case of constructing schedules with tasks with varying runtime requirements; a task $J_i \in G$ with a fixed runtime requirement can be viewed to have the PDF of $\langle(w_{Ji}, 1.0)\rangle$.

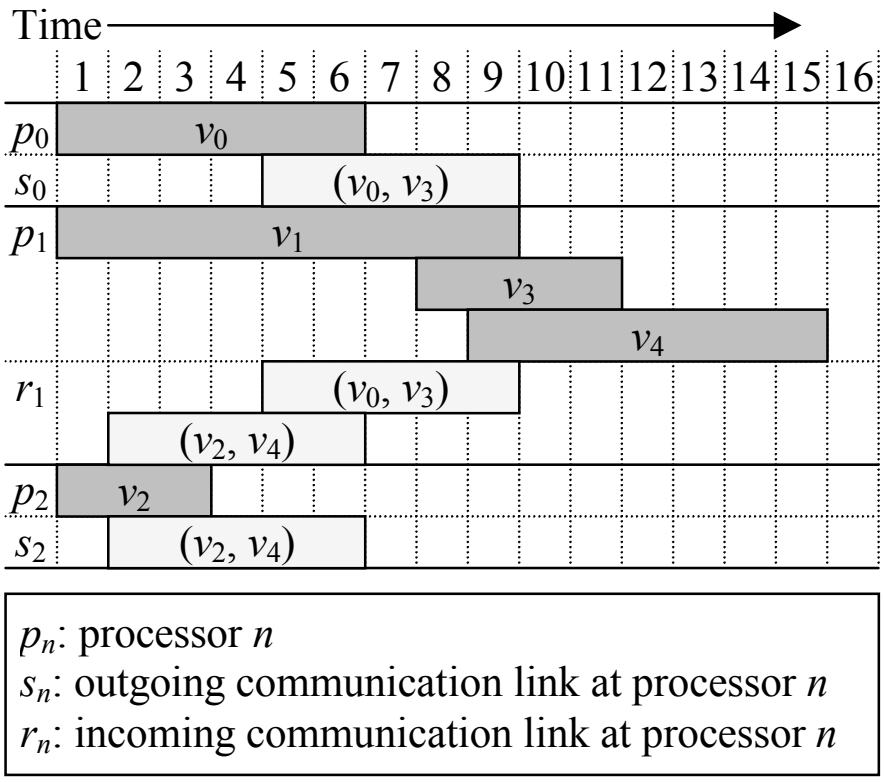


Figure 3.4 Stochastic Schedule with a Slot-Fitting Threshold of 70%

Figure 3.4 shows the Gantt chart for the stochastic schedule for the DAG with variable task weights in Figure 3.2. Note that unlike in the Gantt chart for the deterministic schedule in Figure 3.2, the vertices and edges overlap. This overlap occurs when tasks do not have a 100% probability of utilizing a resource, thereby permitting other tasks to execute if the resource is idle.

Figure 3.5 outlines the algorithm (in pseudocode form) for the selection of the most appropriate processor on which to schedule the highest priority vertex in the ready list. Essentially, the vertex (and all of its immediate predecessor edges) is tentatively scheduled on every processor. The processor that allows the earliest expected start time

for the vertex is permanently allocated to the vertex. Note that after every tentative scheduling action, the resulting start time is noted and the scheduling is reversed in order to enable the scheduling of the vertex on another processor. Reversing a schedule essentially entails reverting the partial schedule to the exact state it was in before the vertex (and the vertex's immediately preceding edges) was added to the schedule.

```
Subroutine schedule_vertex(vertex v)
1.  Let best_processor := NULL
2.  Let best_expected_start_time := ∞

    /* try all processors */
3.  Loop ∀ p : p∈{available processors}
4.          v.start_PDF := call schedule_vertex_on_processor(p, v)
            /* see if the best starting time has been found */
5.          if  (E[v.start_PDF] < best_expected_start_time)
6.                  best_expected_start_time := E[v.start_PDF]
7.                  best_processor := p

            /* reverse any actions taken to schedule the vertex and any preceding
               edges so the schedule on a different processor can be investigated */
8.          call unscheduled_vertex_on_processor(p, v);

    /* commit the schedule on the best processor */
9.  call schedule_vertex_on_processor(best_processor, v);
```

Figure 3.5 Pseudocode Algorithm for Selecting the Best Processor for a Vertex

```
    Subroutine schedule_vertex_on_processor(processor p, vertex v)
        /* initialize the vertex ready time PDF */
1.      Let vertex_ready_PDF := ⟨(0, 1.0)⟩;

        /* process all incoming edges of v, schedule the edge and updating v's ready PDF */
2.      Loop ∀e: e∈{preceding edges of v}
3.              edge_end_PDF := call schedule_edge_on_processor(p, e)
4.              vertex_ready_PDF := max(vertex_ready_PDF, edge_end_PDF)

        /* shift v's ready PDF to the right (it must start after the last edge has completed) */
5.      vertex_ready_PDF := vertex_ready_PDF ⊕ 1

        /* locate the first idle slot in processor p that ends after v's ready time PDF begins
6.      candidate_interval := call find_first_idle_interval(p, vertex_ready_PDF)

        /* keep processing idle slots in p until an appropriate slot is found */
7.      found_interval := false;
8.      loop while (found_interval = false)
                /* the idle slot may begin after the v's ready time */
9.              vertex_start_PDF := max(vertex_ready_PDF, candidate_interval.start_PDF)

                /* compute v's potential completion PDF */
10.             vertex_end_PDF := vertex_start_PDF ⊗ v.weight_PDF

            /* see if v will fit in the idle slot, if not process next idle slot */
11.             if (P(edge_end_PDF ≤ candidate_interval.end_PDF) ≥ slot_fitting_threshold)
12.                     found_interval := true
    13.         else
14.                     candidate_interval := call find_next_idle_interval(p, vertex_ready_PDF)

        /* assign the v to p at the appropriate slot in the schedule */
15.     call insert_vertex(p, candidate_interval, vertex_start_PDF, vertex_end_PDF, v);
16.     if (vertex_end_PDF overlaps candidate_interval.end_PDF)
17.             call ripple_adjust_PDFs(vertex_end_PDF)
```

Figure 3.6 Pseudocode Algorithm for Scheduling a Vertex on a Particular Processor

Figure 3.6 outlines the algorithm for the scheduling of a vertex *v* on a specific processor. Essentially each of the *v*'s immediately preceding edges are scheduled (as outlined in Figure 3.7) and then an idle slot in the processor's schedule is located that can accommodate *v*. Note that *v* can start only after all of the preceding edges and any vertices with start times earlier than *v*'s start time have completed. As each of *v*'s

preceding edges is scheduled, *v*'s ready time PDF is updated (in 4 of Figure 3.6). Lines 6 – 15 in Figure 3.6 describe how a time slot is chosen for executing *v*. After a candidate idle time slot in the processor's schedule is found in which *v* can begin execution, the end time PDF of the candidate slot is compared against the end time PDF of *v*. Vertex *v* is permitted to be scheduled in the candidate slot if the probability that *v* ends before the idle slot ends is at least as much as a pre-specified *slot-fitting threshold* value. The slot-fitting threshold is essentially a real-valued probability parameter that is given to the scheduling procedure, along with the DAG and the processor configuration, at the beginning of the scheduling operation.

Use of the slot-fitting threshold enables the stochastic scheduling algorithms to overlap the completion of one vertex with the starting of another vertex. Without the slot-fitting threshold, one of two following situation occur:

1. Vertex *v* is not overlapped with the next vertex in the schedule. This can result in an underutilization of resources because tasks typically have low probabilities of using resources near the extremes of their starting and ending times.
2. Vertex *v* is forced into the idle slot that is too small to accommodate *v* causing the start time of the next vertex in the schedule to be adjusted to make room for *v*. This forcible insertion of the vertex into a schedule can disrupt the beneficial effects of the heuristic used in the scheduling procedure (*i.e.*, the preferred location of the higher priority ready vertex is lost to a lower priority ready vertex).

Using large values of the slot-fitting threshold (*i.e.*, requiring that the probability of *v* completing before the next vertex in the schedule starts be relatively large) ensures

that the disruption caused by inserting v into the schedule is kept to a minimum. Furthermore, using a slot-fitting threshold of less than 100% allows the scheduling process to better utilize resource. Therefore, the slot-fitting threshold provides another heuristic parameter to control the scheduling process.

After v is inserted into a processor's schedule such that v's execution time overlaps the next vertex, v_{next}, in the processor schedule, the start and end time PDFs of v_{next} must be recomputed in order to account for the overlap. This adjustment leads to a ripple effect of adjustments over the entire partial schedule. Essentially, the start and end time PDFs of all vertices and edges that are topological successors of v_{next} and have already been allocated to the schedule are adjusted. Furthermore, the start and completion time PDFs of even those previously scheduled vertices and edges that are not topological successors of v_{next}, but follow and overlap the topological successors of v_{next} must also be adjusted. This process continues until the start and end time PDFs of all vertices and edges affected by the insertion of v into the schedule are adjusted.

The algorithm for scheduling an edge onto a specific processor is highlighted in Figure 3.7. The procedure is similar to that of scheduling a vertex onto a specific processor except that an edge needs to be simultaneously allocated to two resources: the send and receive links on two different processors. The simultaneous allocation entails looking for idle slots in the schedules of both links whose intersection results in a candidate interval that can accommodate the edge.

```
    Subroutine schedule_edge_on_processor(processor p, edge e)
        /* initialize the edge's ready time PDF to begin after the source vertex ends*/
1.      edge_ready_PDF := e.source_vertex.end_PDF
2.      edge_ready_PDF ⊕ 1
        /* determine the processor on which the originating vertex is scheduled */
3.      p_src := e.source_vertex.scheduled_processor
        /* if the source and destination processors are the same, there is nothing more to do */
        if (p_src = p)
                return edge_ready_PDF

        /* determine the send and recv communication links for the edge */
4.      send_link := source_p.send_link
5.      recv_link := p.recv_link

        /* find the first idle slots in the send & recv links after e's ready time */
6.      send_link_interval := call find_first_idle_interval(send_link, edge_ready_PDF)
7.      recv_link_interval := call find_first_idle_interval(recv_link, edge_ready_PDF);

8.      found_interval := false
        /* keep processing until matching slots are found */
9.      Loop while (found_interval = false)
                /* see if the idle slot can accommodate the edge, otherwise try other idle slots */
10.             candidate_interval.start_PDF := max(edge_ready_PDF,
                                                    send_link_interval.start_PDF,
                                                    recv_link_interval.start_PDF)
11.             candidate_interval.end_PDF := min(send_link_interval.end_PDF,
                                                  recv_link_interval.end_PDF)
12.             edge_end_PDF := candidate_interval.start_PDF ⊗ e.weight_PDF
13.             if (P(edge_end_PDF ≤ candidate_interval.end_PDF) ≥ slot_fitting_threshold)
14.                     found_interval := true;
15.             else if (send_link_interval starts before recv_link_interval)
16.                     send_link_interval := call find_next_idle_interval(send_link,
                                                                           edge_ready_PDF,
                                                                           send_link_interval)
17.             else if (send_link_interval starts after recv_link_interval)
17.                     recv_link_interval := call find_next_idle_interval(recv_link,
                                                                           edge_ready_PDF,
                                                                           recv_link_interval)
19.         else
20.                     send_link_interval := call find_next_idle_interval(send_link,
                                                                           edge_ready_PDF,
                                                                           send_link_interval)
21.             recv_link_interval := call find_next_idle_interval(recv_link, edge_ready_PDF,
                                                                   recv_link_interval)

        /* assign e to the send & recv links at the appropriate processors */
22.     call insert_edge(p_src, send_link, send_link_interval, edge_start_PDF,
                         edge_end_PDF, e)
23.     if (edge_end_PDF overlaps candidate_interval.end_PDF)
24.             ripple_adjust_PDFs(edge_end_PDF)

23.     return edge_end_PDF
```

Figure 3.7 Pseudocode Algorithm for Scheduling an Edge on a Particular Processor

The start time PDF of the candidate interval is computed from the maximum of the start time PDFs of the idle slots in the two links and the edge's ready time PDF. The end time PDF of the candidate interval is computed from the minimum of the end time PDFs of the idle slots in the two links. Next, the edge's proposed completion time PDF is computed by convoluting the candidate interval's start time PDF with the edge's weight PDF. If the probability that the edge completes before the interval ends is greater than the slot-fitting threshold, the edge is inserted into the schedules of the send and receive links. Otherwise a new candidate interval occurring later in the schedules of the two links is located and the procedure is performed again.

If the completion of the inserted edge overlaps the starting times of other previously scheduled edges in either of the two links, rippling adjustment of start and end time PDF recomputations occur, similar to that discussed above in the description of the vertex scheduling procedure. The ripple effect of PDF adjustments is also illustrated in Figure 3.8 and in Table 3.2. In the partial schedule depicted in the figure, vertices v_0, v_1, v_2, and v_3 have already been scheduled along with the edge (v_0, v_3); edge (v_1, v_3) has a negligible weight because v_1 and v_3 are scheduled on the same processor. The table highlights the scheduling steps taken (*i.e.*, steps 1-4(b)) and the resulting PDFs to construct the partial schedule in Figure 3.8.

During the scheduling of vertex v_4, it is determined that process p_1 is the best process to allocate to v_4 because this will eliminate the lengthy edge (v_3, v_4) from the schedule. It is also determined that edge (v_2, v_4) can be scheduled before the previously scheduled edge (v_0, v_3) and will complete before the (v_0, v_3) on the receive link of process

p_1 with a probability of 76.11. Because this probability is greater than the chosen slot-fitting threshold value of 70%, (v_2, v_4) is inserted into the schedule before (v_0, v_3), and this insertion results in the recomputation of the start and end time PDFs of (v_0, v_3) and v_3. After the ripple adjustment is performed, vertex v_4 is scheduled.

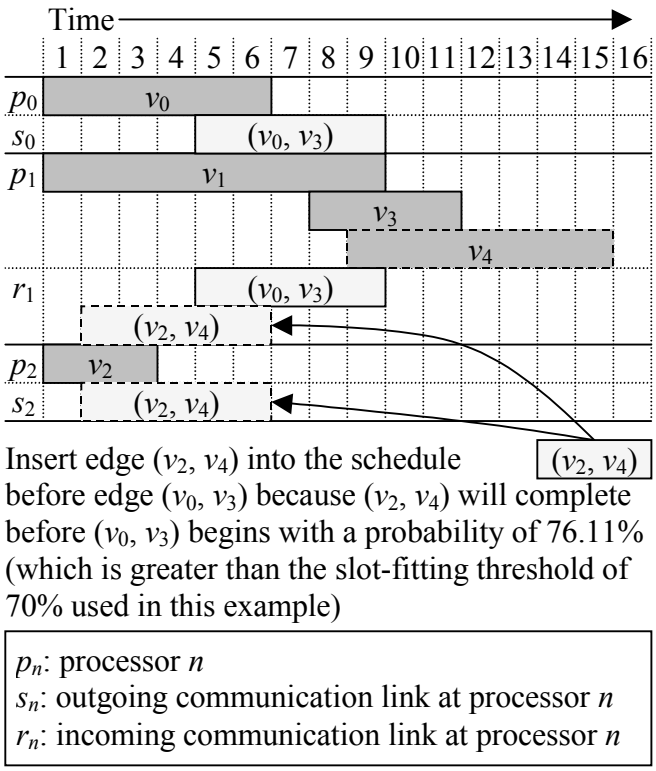


Figure 3.8 Partial Schedule – Scheduling Edge (v_2, v_4) as Part of Scheduling v_4

Table 3.1 Example Sequence of PDF Computations in Stochastic Scheduling

Step 1: schedule v_0 on p_0

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
1	1.0	4	1/3	4	1/3
		5	1/3	5	1/3
		6	1/3	6	1/3

Step 2: schedule v_1 on p_1

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
1	1.0	7	1/3	7	1/3
		8	1/3	8	1/3
		9	1/3	9	1/3

Step 3: schedule v_2 on p_2

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
1	1.0	1	1/3	1	1/3
		2	1/3	2	1/3
		3	1/3	3	1/3

Step 4(a): schedule (v_0, v_3) between p_0 and p_1

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
5	1/3	1	1/3	5	1/9
6	1/3	2	1/3	6	2/9
7	1/3	3	1/3	7	3/9
				8	2/9
				9	1/9

Step 4(b): schedule v_3 on p_1

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
8	0.222	1	1/2	8	0.111
9	0.371	2	1/2	9	0.296
10	0.407			10	0.389
				11	0.204

Step 5(a): schedule (v_2, v_4) between p_2 and p_1

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
2	1/3	1	0.5	2	0.167
3	1/3	2	0.45	3	0.317
4	1/3	3	0.05	4	0.333
				5	0.167
				6	0.016

Step 5(b): adjust (v_0, v_3)

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
5	0.272	1	1/3	5	0.091
6	0.383	2	1/3	6	0.219
7	0.345	3	1/3	7	0.333
				8	0.243
				9	0.114

Step 5(c): adjust v_3

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
8	0.214	1	1/2	8	0.107
9	0.376	2	1/2	9	0.295
10	0.410			10	0.393
				11	0.205

Step 5(d): schedule v_4 on p_1

Start PDF		Wt. PDF		End PDF	
Time	Prob.	Time	Prob.	Time	Prob.
9	0.107	2	1/3	10	0.036
10	0.295	3	1/3	11	0.134
11	0.393	4	1/3	12	0.265
12	0.205			13	0.298
				14	0.199
				15	0.068

Table 3.2 Figure 3.9 depicts the Gantt chart that results when a slot-fitting threshold greater than 76.11% (*e.g.*, 100%) is used. In this case, edge (v_2, v_4) cannot be inserted into the schedule before edge (v_0, v_3) and must, instead, be scheduled to begin at time unit 6. This delay in the execution of (v_2, v_4) causes a delay in the starting and completion of v_4 and a consequent increase in the schedule length as compared to the schedule produced with a more liberal slot-fitting threshold value.

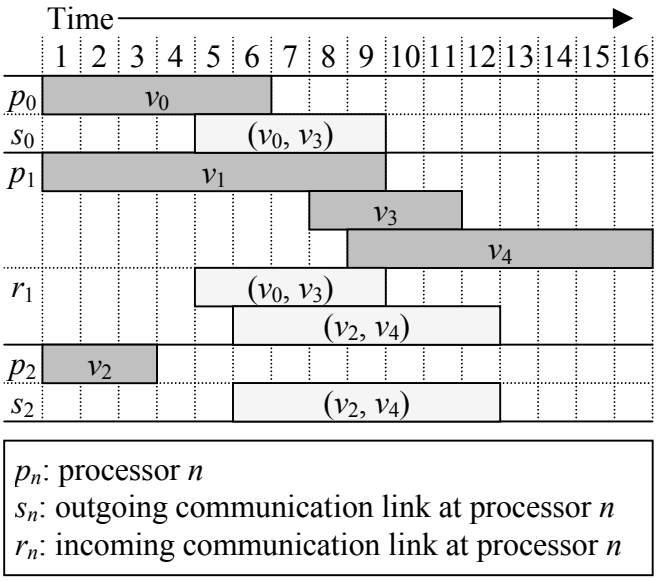


Figure 3.9 Stochastic Schedule with a Slot-Fitting Threshold of 100%

3.7 Genetic List Scheduling Approach

The genetic list scheduling (GLS) algorithm developed in this dissertation uses a GA to determine the order in which ready nodes are processed by the LS algorithm. The GA techniques described below are adapted from a variety of sources in order to

construct stochastic schedules. The original contribution of this section is the use of stochastic resource allocation techniques for evaluating chromosomes in the GA.

The outline of the steady state GLS algorithm investigated in this dissertation is described in Figure 3.10 below.

1. Generate the initial population.
2. Use list scheduling to construct a schedule in order to evaluate each of the initial chromosomes.
3. Loop while the termination criteria are not satisfied:
    4. Select a genetic operator.
    5. Select chromosome(s) from the local population and apply the operator to produce the offspring chromosome.
    6. Use LS to construct a schedule in order to evaluate the offspring chromosome.
    7. Select chromosome from the local population to be replaced by the offspring chromosome.
8. Use the fittest chromosome to construct the solution schedule.

Figure 3.10 The Fundamental GLS Algorithm

Each chromosome in the GLS developed in this research consists of two vectors of genes. The *vertex vector* contains a gene for each vertex in the DAG and the *edge vector* contains a gene for each edge in the DAG. Essentially, each gene uniquely identifies its corresponding vertex or edge and there are $|V| + |E|$ genes in each chromosome. The position of the vertex and edge genes in their respective vectors determines the priority of the corresponding vertices and edges used by the list scheduler. The use of both vertices and edges in the chromosome enables the GA to optimize both vertex and edge scheduling.

Two different crossover operators, *ordered crossover* (OX) [42] and *vector crossover* (VX) are used in this GSL algorithm. In OX, a single crossover point is randomly selected in the vertex vector. The sequence of genes prior to the crossover point is copied from the first parent to the front of the vertex vector in the offspring chromosome. The remaining genes in the first parent (*i.e.*, following the crossover point) are copied into the offspring gene in the order they appear in the second parent. The same operation is also performed on the edge vectors to construct the offspring chromosome. In VX, the first parent contributes a complete copy of its vertex vector and the second parent contributes a complete copy of its edge vector to construct the offspring chromosome.

The mutation operator swaps the location of a pair of genes within the vertex vector, the edge vector, or a pair each from both vertex and edge vectors. One of these three options is selected with equal probability. Experimentation is required to select the optimal probabilities of selecting the various crossover and mutation operators at each GA step. Initial empirical results suggest that this GLS produces good results when the probabilities of selecting the OX, VX, and mutation operators are $\pi_{ox} = 0.75$, $\pi_{vx} = 0.20$, and $\pi_m = 0.05$, respectively.

Other researchers (*e.g.*, [13]) have observed that making direct use of the objective function (*i.e.*, schedule length in this case) to compute the fitness of chromosomes can lead to premature convergence. This occurs when a single chromosome with a significantly better than average objective function value is repeatedly selected for breeding, leading to reduced genetic diversity in the population.

In order to prevent this situation, the rank of chromosomes, rather than their objective function values are used to compute their fitness. The rank of chromosome *c* is the number of chromosomes in population Ω that produce worse schedules than *c*.

In order to further reduce the ability of high-ranking individuals to dominate the population, Grajcar proposed in [67], that the fitness of a chromosome be made inversely proportional to the number of offspring it has produced. This results in the following fitness function for chromosome *c*:

$$\varphi(c) = \frac{rank(c)}{|offspring(c)|+1}, \tag{3.1}$$

where *offspring*(*c*) is the number of offspring produced by chromosome *c* to date.

The chromosome with the largest fitness value in a random subset of Ω is selected for reproduction. Similarly, the chromosome with the least fitness value from another random subset of Ω is selected for replacement. In early experiments, selection subset sizes ranging from 2% to 10% of |Ω| worked well for populations ranging from 200 to 800 chromosomes.

A parallel implementation of the GLS using the *synchronous connected island model* [64] is used in this dissertation. In this implementation, 15 parallel GLS processes operate on separate local populations and communicate synchronously to exchange the fittest chromosomes with each other periodically. Essentially, each of the 15 processes synchronously transmits the best chromosome in its local populations to all the other processes. Each process then adds the best chromosome received into its local population. This "migration" of chromosomes enhances the exploitation of high quality

solutions in parallel GAs, while the relative isolation of the subpopulation enhances genetic diversity and prevents premature convergence.

In the parallel GLS implementation used for this dissertation, the number of iterations between migrations is as follows: 4,000, 4,000, 4,000, 4,000, 2,000, 1,000, 500, 250, 250, 125, 125, …, 125. This strategy emphasizes exploration of the solution space at the beginning of the evolutionary process and emphasizes exploitation of good genetic information towards the latter stages. A total of 24,000 iterations are performed in each process and the local population size is 1,000 chromosomes. The fittest chromosome from a pool of 50 randomly selected chromosomes is used for participation in the crossover and mutation operators. The worst chromosome from a different pool of randomly selected chromosomes is discarded and replaced by the new chromosome resulting from the crossover or mutation operator. The local population size remains at 1,000 chromosomes.

As suggested by Potts, Giddens, and Yadav [127], varying the GA control parameters in each of the N parallel GLS processes can lead to increased genetic diversity. Therefore, the mutation, vector crossover, and ordered crossover rates used in a parallel GLS process $n_i \in \{0, 1, \ldots, N - 1\}$ are given by the following:

$$\pi_m(n_i) = 0.05 + 0.05 \times n_i/N, \tag{3.2}$$

$$\pi_{vx}(n_i) = 0.20 + 0.10 \times n_i/N, \text{ and} \tag{3.3}$$

$$\pi_{xo}(n_i) = 0.75 - 0.15 \times n_i/N, \tag{3.4}$$

respectively. This implies that the mutation rates range between 0.50 and 0.10, the feature crossover rates range between 0.20 and 0.30, and the order crossover rates vary

between 0.75 and 0.60. These probability ranges for the recombination operators worked well in the deterministic GLS scheduling experiments reported in [41], and therefore, have been adapted for the GLS experiments conducted as part of this dissertation.

3.8 Scheduling Options

Two methods for constructing schedules can be used for each of the three distinct LS approaches and the GLS approach developed in this dissertation. The first method, designated as the *exact method*, is to directly utilize the weight PDFs as described in detail in the previous sections of this chapter to determine the starting time and end time PDFs for all tasks, and to use those PDFs in making scheduling decisions. The exact approach is the primary focus of this research. However, the repeated PDF computations (*i.e.*, convolution, minimum, and maximum) can be time consuming as evidenced in Chapter V.

In order to reduce the time taken to construct schedules, the second option, suggested by Dr. Eric Hansen (dissertation committee member) in private communication, is to use a fixed estimate of the weight for each task during schedule construction. This method is designated as the *estimate method*. The estimate method essentially reduces the number of points in every task's PDF to one (*i.e.*, task J_i's the PDF essentially becomes $\langle(E_i, 1.0)\rangle$: $l_{wi} \leq E_i \leq u_{wi}$, where E_i is the estimate value of J_i's weight PDF). Note that using an estimated value $E_i = u_{wi}$ for all $J_i \in G$ results in a WCET schedule. Similarly, using an estimated value of $E_i = l_{wi}$ results in best case execution

time schedule and using the expected values for estimated values will result in a schedule using the average task weights.

Clearly, schedules using WCET will result in resource utilization and reduced performance for soft real-time applications that can tolerate occasional missed deadlines. Conversely, schedules using the best-case execution time will only rarely meet deadlines because the probability that all tasks take the minimum amount of time to execute is relatively small. Using average execution time estimate or some other estimate (*e.g.*, 99% of WCET) also does not provide an accurate means for establishing the probability with which the schedule will meet its deadline. It is also difficult to predict what percent of WCET should be used as an estimate in order to construct schedules that provide a required probability of meeting deadlines.

In order to overcome these problems with applying deterministic schedules to soft real-time applications requiring statistical guarantees, an additional step in the scheduling process is used to construct accurate start and completion time PDFs from the deterministic schedule. The purpose of the deterministic schedule is to establish a resource allocation and a strict ordering between the tasks. This resource allocation and ordering is used in the second stage to compute the start time and completion time of each task using the convolution and maximum PDF operations.

The use of the estimate method to construct stochastic schedules is potentially faster than the exact method because the construction of the initial deterministic schedule avoids the computation of the numerous start time and completion time PDFs that must be performed while the scheduling routines look to the best slot in the best resource for

each task. In the estimate method, the start time and completion time PDFs are computed only once for each task. However, the estimate method is unable to take advantage of the slot-fitting technique available to the exact method. This is because in the estimate method, a task must fit in a candidate slot; there is no technique available to accurately determine the probability that the inserted task will complete before the end of the candidate slot. The advantages and disadvantages of the two methods are empirically evaluated in Chapter V.

3.9 Reducing Stochastic Jitter

Consider a stochastic schedule in which a set of n tasks $\{J_1, J_2, \ldots, J_n\}$ are scheduled one after another in sequence such that task J_{i+1} is scheduled to execute as soon as task J_i completes. Let $[l_{s0}, u_{s0}]$ be the interval over which the start time PDF of J_1 is defined. Also, let $[l_{wi}, u_{wi}]$, where $1 \leq i \leq n$, be the interval over which the weight of task J_i is defined. According to Lemma 1, task J_1's completion time PDF is defined over the interval $[l_{s0} + l_{w1} - 1, u_{s0} + u_{w1} - 1]$. Because task J_2 begins immediately after J_1 completes, the start time PDF of J_2 is computed by translating J_1's completion time PDF by one time unit to the right. This results in $[l_{s0} + l_{w1}, u_{s0} + u_{w1}]$ as the interval over which J_2's start time PDF is defined. After convoluting J_2's start time PDF with J_2's weight PDF, J_2's end time PDF is defined over the interval $[l_{s0} + l_{w1} + l_{w2} - 1, u_{s0} + u_{w1} + u_{w2} - 1]$. After this process continues for all n tasks, the final end time PDF of the sequence of tasks is given by the following expression:

$$[l_{fn}, u_{fn}] = [l_{s0} + l_{w1} + \ldots + l_{wn} - 1, u_{s0} + u_{w1} + \ldots + u_{wn} - 1] \quad (3.5)$$

The difference between u_{fn} and l_{fn} specified in equation (3.5) gives the jitter in completion time for the sequence of tasks as follows:

$$\begin{aligned}\delta &= u - l \\ &= u_{s0} + u_{w1} + \cdots + u_{wn} - 1 - (l_{s0} + l_{w1} + \cdots + l_{wn} - 1) \\ &= u_{s0} + u_{w1} + \cdots + u_{wn} + 1 - l_{s0} - l_{w1} - \cdots - l_{wn} + 1) \\ &= u_{s0} - l_{s0} + u_{w1} - l_{w1} + \cdots + u_{wn} - l_{wn}\end{aligned} \tag{3.6}$$

From equation (3.6) it is clear that the completion time jitter of the completion time of the sequence of tasks is the sum of the jitter in execution time requirements of each task in the sequence. This implies that the long sequences of tasks in schedule will result in a significant amount of jitter introduced by the stochastic nature of the tasks. This jitter is referred to as the *stochastic jitter* in this dissertation.

The presence of task completion jitter is undesirable in many real-time control systems [28, 121], and therefore, a technique for systematically reducing the stochastic jitter resulting from stochastic scheduling of DAGs is proposed in this section.

The stochastic jitter for task J_i in the sequence of tasks from the previous discussion above is given by the following expression:

$$\delta_i = u_{s0} - l_{s0} + u_{w1} - l_{w2} + \ldots + u_{wi} - l_{wi}. \tag{3.7}$$

In equation (3.7), $u_{wi} - l_{wi}$ is the contribution of the inherent variability of the execution time requirements of J_i to the overall completion time jitter of J_i. However, $u_{s0} - l_{s0} + u_{w1} - l_{w2} + \ldots + u_{wi-1} - l_{wi-1}$ is the contribution in jitter caused by the earlier tasks. Clearly, the stochastic jitter in the completion of J_i can be reduced if instead of starting as soon as possible, the start time of J_i is delayed by some period of time. In

order to be useful, this delay must be bounded such that J_i still completes no later that the latest possible time that would have resulted had the start of J_i not been delayed.

From the discussion leading to equation (3.5) above, the interval of the start time PDF of J_i is $[l_{si}, u_{si}] = [l_{s0} + l_{w1} + \ldots + l_{wi-1}, u_{s0} + u_{w1} + \ldots + u_{wi-1}]$ and the interval of the completion time PDF of Ji is $[l_{fi}, u_{fi}] = [l_{s0} + l_{w1} + \ldots + l_{wi} - 1, u_{s0} + u_{w1} + \ldots + u_{wi} - 1]$. Note that $u_{fi} = u_{si} + u_{wi} - 1$. Therefore, as long as J_i begins no later than time u_{si}, J_i will always complete at or before time u_{fi} because J_i can never take more the u_{wi} time units to execute.

Theorem 4 provides a technique for computing the new start time PDF of a delayed task given the original start time PDF and a delay quantity. This technique can be used to reduce the completion jitter of all the tasks in the schedule. However, the tradeoff is that a greater proportion of the task execution burden is placed on the later portions of the overall schedule. This implies that the probability of missing deadlines when the schedule length is reduced is greater in the schedules with reduced stochastic jitter as compared to the probability of missing deadlines in the schedule with no jitter control.

3.10 Complexity Analysis

This section presents an analysis of the complexity of the PDF manipulation operators and the scheduling algorithms presented in this chapter.

3.10.1 Complexity of PDF Operators

The convolution operator, as implemented in this research is of $O(n^2)$ complexity [61], where n is the average width of each of the PDFs in the convolution. It is possible to implement an $O(n\log_2 n)$ complexity PDF, as shown in [61]. However, this reduction in complexity also results in reduced accuracy.

Lemma 5: *The PDF resulting from the convolution of two PDFs of width n has width (2n-1)*.

Proof: Assume the two PDFs A and B are defined within the intervals $[l_A, l_A + n - 1]$ and $[l_B, l_B + n - 1]$, respectively. According to Lemma 1, the lower bound of the PDF resulting from the convolution of A and B is $l_A + l_B - 1$. The upper bound of the resulting PDF is given by the following:

$$\begin{aligned} l_{A\otimes B} &= l_A + n - 1 + l_b + n - 1 - 1 \\ &= l_A + l_b + 2n - 3 \end{aligned} . \tag{3.8}$$

The width of the resulting interval is given by the following:

$$\begin{aligned} width &= u_{A\otimes B} - l_{A\otimes B} + 1 \\ &= l_A + l_B + 2n - 3 - (l_A + l_B - 1) + 1 \\ &= l_A + l_B + 2n - 3 - l_A - l_B + 1 + 1 \\ &= 2n - 1 \end{aligned} . \tag{3.9}$$

Computing the maximum PDF from k PDFs of width n has complexity of $O(n)$. From Figure 3.5, it is evident that line 7 is executed n times for each of the $k - 1$ PDFs after the first PDF is used to initialize the resulting PDF. Similarly, computing the minimum PDF also has a complexity of $O(n)$.

3.10.2 Complexity of the Exact Method List Scheduling Algorithms

There are two primary factors that influence the amount of time taken by the algorithms. The first factor is the average width of the weight PDFs of the tasks in the DAG. Manipulation of wider PDFs requires more time to compute than the manipulation of narrower PDFs. The second factor is the number of tasks in the DAG; increasing the number of tasks increases the amount of time required to construct the schedules. This section will focus on developing the complexity characteristics of the scheduling algorithms based on these two parameters.

Two DAG structures can be used to analyze the complexity of the LS algorithms presented in this chapter. The first DAG structure is the *Linear DAG* in which the vertices are arranged in linear order with each successive vertex connected to a single predecessor vertex by an edge. The second DAG structure is the *Unordered DAG* in which there are no edges (*i.e.*, there are no precedence constraints between vertices and all vertices can be executed simultaneously). The vertices in the linear structure DAG are scheduled to execute on a single processor sequentially, causing the completion time PDFs of successive vertices to grow rapidly, while keeping the number of processors examined is kept to a minimum because essentially only the processor on which the previous vertex is scheduled and an idle processor have to be examined in order to determine the processor that will allow the vertex to be scheduled as early as possible (note that the vertex will begin the earliest when scheduled to execute on the processor on which the preceding vertex is scheduled). The unordered DAG enables the vertices to be scheduled on any processor but only an idle processor will allow the earliest start time.

This results in an increased amount of search while minimizing the width of the resulting PDFs because only single vertices are scheduled on any processor, assuming an unlimited number of available processors.

Theorem 6: *The complexity of the Exact SETF algorithm for the linear DAG is $O(n^2v^2)$ where v is the number of vertices in the DAG and n is the average width of the intervals over which the weight PDFs of the vertices are defined.*

Proof: Table 3.3 presents a summary of the floating point operations that are performed during the scheduling process. In step 1, there is only one ready vertex and all processors are idle. The ready vertex is scheduled on an idle processor and the completion time PDF of this vertex is determined by the convolution of the width of the ready vertex's weight PDF with the start time PDF of $\langle(1, 1.0)\rangle$. This convolution results in n operations in step 1.

Table 3.3 Summary of Operations in SETF for the Linear DAG

Step	No. of Ready vertices	No. of Processors with Scheduled Vertices	Width of Completion Time PDF of Last Vertex on Processor	Total Number of Operations on Non-idle Processors	Number of Operations on Idle Processor
1	1	0	0	0	n
2	1	1	n	$n(n-0)$	n
3	1	1	$2n-1$	$n(2n-1)$	n
…	…	…	…	…	…
v	1	1	$(v-1)n-(v-2)$	$n[(v-1)n-(v-2)]$	n

In step 2, there is one ready vertex and the algorithm examines the scheduling of the ready vertex on the processor with the previously scheduled vertex and an idle

processor. The ready vertex is finally scheduled on the processor with the previously scheduled vertex because this eliminates the edge weight between the ready vertex and the previous vertex. The completion time PDF of the previously scheduled vertex is of width n. The start time PDF of the ready vertex is a translated completion time PDF of the previous vertex. The Convolution of the start time PDF with the width PDF of the ready vertex results in n^2 operations and the completion time PDF resulting from the convolution has a width of $2n - 1$. The tentative scheduling of the vertex on an idle processor requires n operations, as explained in step 1.

In step 3, there is again a single ready vertex that will be tentatively scheduled on the processor with the previous two vertices and an idle vertex. As before, the ready vertex is finally scheduled on the processor with the previous vertices, and this requires $n(2n - 1)$ operations. The tentative scheduling of the vertex on an idle processor requires n operations, as explained in steps 1 and 2.

This process continues until all v vertices have been scheduled. Summing the total number of operations results in the following complexity upper bound:

$$\begin{aligned} n + \sum_{i=1}^{v-1} n[in - (i-1)] + n &= n + \sum_{i=1}^{v-1} n^2 i - in + 2n \\ &= n + (n^2 - n)\sum_{i=1}^{v-1} i + 2n\sum_{i=1}^{v-1} 1 \\ &= n + \frac{(n^2 - n)v(v-1)}{2} + 2n(v-1) \\ &= n + \frac{n^2v^2 - n^2v - nv^2 + nv}{2} - 2nv - 2n \\ &= O(n^2v^2) \end{aligned} \tag{3.10}$$

This completes the proof for theorem 6. □

Theorem 7: *The complexity of the Exact SETF algorithm for the unordered DAG is $O(n^2v^3)$ where v is the number of vertices in the DAG and n is the average width of the intervals over which the weight PDFs of the vertices are defined.*

Proof: Table 3.4 presents a summary of the floating point operations that are performed during the scheduling process. In step 1, all vertices are ready and all processors are idle. Each ready vertex is tentatively scheduled on an idle processor. The completion time PDFs of these vertices are determined by the convolution of the width of the vertices weight PDFs with the start time PDF of $\langle(1, 1.0)\rangle$. This convolution results in n operations for each of the v vertices in step 1. One of the ready vertices is arbitrarily selected to be scheduled on an arbitrarily selected idle processor.

Table 3.4 Summary of Operations in SETF for the Unordered DAG

Step	No. of Ready vertices	No. of Processors with Scheduled Vertices	Width of Completion Time PDF of All Scheduled Vertices	Total Number of Operations on Non-idle Processors	Number of Operations on Idle Processor
1	v	0	0	0	$(v-0)n$
2	$v-1$	1	n	$(v-1)1n^2$	$(v-1)n$
3	$v-2$	2	n	$(v-2)2n^2$	$(v-2)n$
…	…	…	…	…	…
v	1	$v-1$	n	$1(v-1)n^2$	$1n$

In step 2, there are $(v - 1)$ ready vertices and the algorithm examines the scheduling of all the ready vertices on the processor with the previously scheduled vertex and an idle processor. An arbitrary ready vertex is finally scheduled on an arbitrary idle processor. The completion time PDF of the previously scheduled vertex is of width n and

the start time PDF of a ready vertex scheduled to follow the previously scheduled vertex is a translated completion time PDF of the previously scheduled vertex. The Convolution of the start time PDF with the width PDF of each of the ready vertices results in n^2 operations and the completion time PDF resulting from the convolution has a width of $2n - 1$. The tentative scheduling of the vertices on an idle processor requires n operations for each vertex.

In step 3, there are $(v - 2)$ ready vertices that will be tentatively scheduled on the two processor with the previous two vertices and an idle vertex. As before, an arbitrary ready vertex is finally scheduled on an arbitrary idle processor. This process continues until all v vertices have been scheduled. Summing the total number of operations results in the following complexity upper bound:

$$
\begin{aligned}
&\sum_{i=0}^{v-1}(v-i)in^2+(v-i)n \\
&=\sum_{i=0}^{v-1}vin^2-i^2n^2+vn-in \\
&=vn^2\sum_{i=0}^{v-1}i-n^2\sum_{i=0}^{v-1}i^2+vn\sum_{i=0}^{v-1}1-n\sum_{i=0}^{v-1}i \\
&=vn^2\frac{(v-1)v}{2}-n^2\frac{(v-1)(v-1+1)[2(v-1)+1]}{6}+vn(v-1)+n\frac{(v-1)v}{2} \\
&=\frac{v^3n^2-v^2n^2}{2}-\frac{2n^2v^3-n^2v^2-2nv^2+nv}{6}+v^2n-vn-\frac{nv^2-nv}{2} \\
&=\frac{3v^3n^2-3v^2n^2-2n^2v^3+n^2v^2+2nv^2-nv}{6}+v^2n-vn-\frac{nv^2-nv}{2} \\
&=\frac{v^3n^2-2v^2n^2+2nv^2-nv}{6}+v^2n-vn-\frac{nv^2-nv}{2} \\
&=O(n^2v^3)
\end{aligned}
\tag{3.11}
$$

This completes the proof for theorem 7. □

Theorem 8: *The complexity of the Exact SHLEFT and Exact SCP algorithms for the linear DAG is $O(n^2v^2)$ where v is the number of vertices in the DAG and n is the average width of the intervals over which the weight PDFs of the vertices are defined*.

Proof: The scheduling steps in the Exact SHLEFT and Exact SCP algorithms for the linear DAG follow the same pattern as exhibited by the Exact SETF algorithm for the linear DAG. At each step there is exactly one ready vertex. All vertices are scheduled on the same processor and the algorithms tentatively schedule ready vertices on the processor with the previous vertices and an idle processor. Therefore, the complexity of the scheduling steps in the Exact SHLEFT and Exact SCP algorithms is the same as the complexity of the scheduling steps in the Exact SETF algorithm of $O(n^2v^2)$.

Unlike Exact SETF however, the Exact SHLEFT and Exact SCP algorithms require a preprocessing step in order to determine each vertex's scheduling priority. Exact SHLEFT requires the computation of the stochastic b-level attribute for each vertex. Computing the b-level for all vertices in the DAG is linear in terms of the number of vertices and edges in the DAG. Therefore, the complexity of the scheduling steps dominates the complexity of the b-level computation. This makes the overall complexity of the Exact SHLEFT algorithm $O(n^2v^2)$.

The vertex priority assignment algorithm in Exact SCP algorithm is a two pass algorithm. The forward pass uses a simplified version of the scheduling steps, and therefore, has complexity of $O(n^2v^2)$. The backwards pass essentially reverses the scheduling steps and also has complexity of $O(n^2v^2)$. This makes the overall complexity of the Exact SCP algorithm $O(n^2v^2)$. □

Theorem 9: *The complexity of the Exact SHLEFT and Exact SCP algorithms for the unordered DAG is $O(n^2v^2)$ where v is the number of vertices in the DAG and n is the average width of the intervals over which the weight PDFs of the vertices are defined.*

Proof: Table 3.5 presents a summary of the floating point operations that are performed during the scheduling process. Note that all vertices are ready to be scheduled simultaneously. In step 1, the highest priority ready vertex is scheduled on an idle processor and the completion time PDF of this vertex is determined by the convolution of the width of the ready vertex's weight PDF with the start time PDF of $\langle(1, 1.0)\rangle$. This convolution results in n operations in step 1.

Table 3.5 Summary of Operations in SHLEFT and SCP for the Unordered DAG

Step	No. of Ready vertices	No. of Processors with Scheduled Vertices	Width of Completion Time PDF of All Scheduled Vertices	Total Number of Operations on Non-idle Processors	Number of Operations on Idle Processor
1	1	0	0	0	n
2	1	1	N	n^2	n
3	1	2	N	$2n^2$	n
…	…	…	…	…	…
v	1	$(v-1)$	N	$(v-1)n^2$	n

In step 2, there are $(v - 1)$ ready vertices remaining and the algorithm examines the scheduling the highest priority ready vertex on the processor with the previously scheduled vertex and an idle processor. The highest priority ready vertex is finally scheduled on an idle processor. The completion time PDF of the previously scheduled vertex is of width n and the start time PDF of a ready vertex scheduled to follow the

previously scheduled vertex is a translated completion time PDF of the previously scheduled vertex. The Convolution of the start time PDF with the width PDF of each of the ready vertices results in n^2 operations. The tentative scheduling of the vertices on an idle processor requires n operations for each vertex.

In step 3, there are $(v - 2)$ ready vertices and of these the vertex with the highest priority will be tentatively scheduled on the two processor with the previous two vertices and an idle vertex. As before, the highest priority ready vertex is finally scheduled on an idle processor. This process continues until all v vertices have been scheduled. Summing the total number of operations results in the following complexity upper bound:

$$\begin{aligned}\sum_{i=0}^{v-1} in^2 + n &= n^2 \sum_{i=0}^{v-1} i + n \sum_{i=0}^{v-1} 1 \\ &= n^2 \frac{(v-1)v}{2} + n(v-1) \\ &= \frac{n^2(v^2 - v)}{2} + nv - n \\ &= \frac{n^2v^2 - n^2v}{2} + nv - n \\ &= O(n^2v^2)\end{aligned} \tag{3.12}$$

The priority assignment step in SHLEFT is linear in terms of the number of vertices in the DAG, this makes the complexity of the exact SHLEFT algorithm $O(n^2v^2)$. Because there are no precedence relationships between the vertices in the unordered DAG, the forward and backwards passes in the SCP algorithm have a complexity of $O(n^2)$ for each vertex, resulting in an overall priority assignment complexity of $O(n^2v)$. Therefore, the complexity of the Exact SCP algorithm is also $O(n^2v^2)$. □

CHAPTER IV

EXPERIMENT DESIGN

A number of stochastic schedules are created for a variety of directed acyclic graphs (DAGs) in order to evaluate the veracity, applicability, benefits, and costs of the probability distribution functions (PDF) manipulation operations developed in Chapter III. The empirical results are also used to test the effectiveness of the stochastic scheduling techniques developed in Chapter III in reducing schedule lengths and reducing jitter at the cost of increased probability of missing end-to-end deadlines. This chapter describes the experiments that were conducted as part of this dissertation, presents experimental data, and provides an analysis of the results.

4.1 Directed Acyclic Graph Classes

In order to support the hypothesis, schedules for a variety of DAGs with varying structures, sizes, weight distributions, and communication vs. computation requirements are created and analyzed. These DAGs present a wide range of scheduling problems to the stochastic scheduling techniques developed in this dissertation. Analysis of the schedules produced by these techniques for a variety of scheduling problems provide insights into the ability of the various novel stochastic scheduling heuristics and schedule control parameters to influence the quality of service (QoS) and performance

characteristics of schedules. This section describes these DAG characteristics in more detail.

4.1.1 DAG Structure

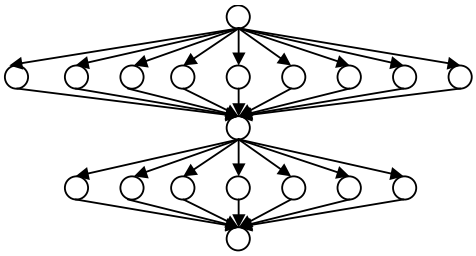
Each of the following six distinct structural forms were used to create several DAGs:

1. *Fast Fourier Transform* (FFT),
2. *Hierarchical Fork-Join* (HFJ)
3. *Mean Value Analysis* (MVA),
4. *Out Tree* (OUT),
5. *Random*, and
6. *Simple Fork-Join* (SFJ).

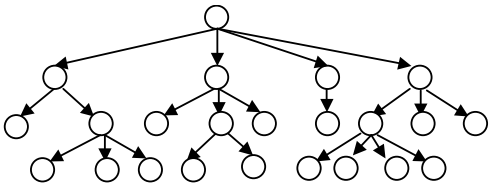
These structure types were selected because of their pedagogically interesting features as well as to provide DAGs for a representative set of realistic parallel applications. Similar DAG structures have also been used in evaluating list scheduling heuristics developed and applied for scheduling tasks with fixed execution time requirements [92, 93].

The HFJ and SFJ DAG structures, illustrated in Figure 4.1(a) and Figure 4.1(e), emphasize the branching and joining data flows that have largely been ignored by the existing probabilistic scheduling research, where interest has focused primarily on periodic scheduling. These structures also represent the class of trivially parallel tasks wherein a single task performs sequential preprocessing and distributes the workload

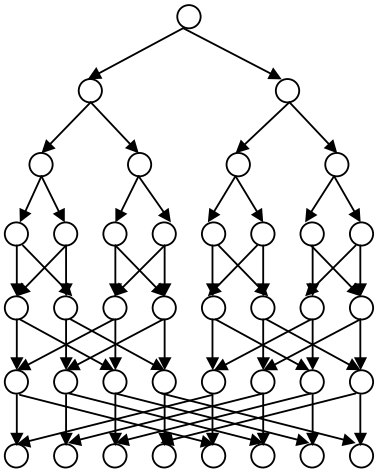
between several parallel tasks, and then, another sequential task gathers the results from the parallel tasks.



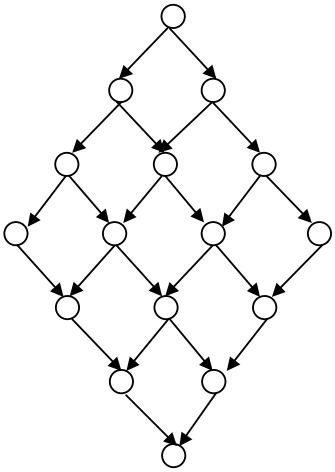
(a) Simple Fork-Join



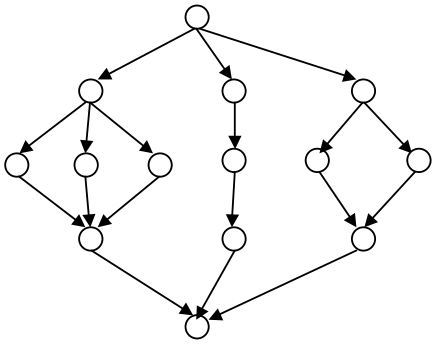
(b) Out Tree



(c) Fast Fourier Transform



(d) Mean Value Analysis



(e) Hierarchical Fork Join

Figure 4.1 Miniature Examples of DAG Structures

The FFT DAG structure, illustrated in Figure 4.1(c), represents a commonly used parallel task performed in many real-time image and signal processing applications. The MVA structure, illustrated in Figure 4.1(d), represents the structure of a parallel application and also provides a pedagogically interesting structure with several branching and joining flows. The OUT DAG structure, illustrated in Figure 4.1(b), is interesting because it exercises the ability of the scheduling techniques to effectively schedule a large number of data and flow control branches. The random DAG structure presents a significantly more irregularity in form as compared to the other structures.

4.1.2 Communication to Computation Ratio

Definition 8: The *computation-to-communication ratio* (CCR) for a DAG is defined as the ratio of average expected vertex weight to the average expected edge weight. Formally, the CCR of DAG, G, is computed as follows:

$$CCR(G) = \frac{\frac{\sum_{v_i \in V} E[w(v_i)]}{|V|}}{\frac{\sum_{e_i \in E} E[w(e_i)]}{|E|}} \tag{4.1}$$

In order to analyze the ability of the new scheduling mechanisms to handle applications with a variety of CCRs, DAGs with the following CCRs are constructed:

1. DAGs with CCR = 0.5 represents applications whose communication tasks take 100% more time as compared to computation tasks, on average (*i.e.*, communication tasks take twice long as computation tasks).

2. DAGs with CCR = 0.67 represent applications whose communication tasks take 50% more time as compared to computation tasks, on average.
3. DAGS with CCR = 1.0 represent applications whose communication and computation tasks take the same amount of time, on average.
4. DAGs with CCR = 1.5 represent applications whose computation tasks take 50% more time as compared to communication tasks, on average.
5. DAGS with CCR = 2.0 represent application whose computation tasks take 100% more time as compared to communication tasks, on average.

This range of choices should provide an indication of what influence, if any, CCR has on the ability of the schedulers to trade probability of missing deadlines for shorter schedules.

4.1.3 Task Weight Probability Distributions

Assigning weight probability distributions to a task in a DAG is a three-step process. In the first step, the required expected weight of the task is selected from a normally distributed random variable. The expected value of the task's weight distribution is established first in order to remain within the CCR constraints of the particular DAG. Next, the task's weight random probability distribution is generated such that the number of distinct domain points in the PDF is 15% of the expected weight value. Specifically, the weight distribution, π_w, is defined over the following interval:

$$[l_{\pi w}, u_{\pi w}] = [1, 0.15W], \tag{4.2}$$

where W is the required expected weight of the task.

Next, the PDF domain interval is translated in order to result in a PDF with the required expected value as follows:

$$[l_w, u_w] = [W - \lceil E[\pi_w] \rceil + 1, W - \lceil E[\pi_w] \rceil + u_{\pi w}], \tag{4.3}$$

where $E[\pi_w]$ is the expected weight of the original PDF π_w. For example, let the required expected weight of a task be 200 time units. A weight PDF defined over the interval [1, 30] is generated initially. Let the PDF have an expected value of 9.5. The domain of this PDF is translated by x units such that the PDF is now defined over the interval [190, 220] (*i.e.*, [200 – 10 + 1, 200 – 10 + 30]).

In order to test the efficacy of the PDF operations in the stochastic scheduling techniques developed in this dissertation, several DAGs using three distinct probability distributions were created.

The first type of distribution used is the Beta distribution [45]. Beta distributions are well suited for modeling real-time applications because these distributions are defined (*i.e.*, have positive non-zero values) only over a finite interval [*l*, *u*]. This restriction resembles the behavior of real-time tasks that are designed to complete within a relatively narrow range of times.

The beta probability distributions are computed using following equation:

$$\pi(x) = \frac{(x-l)^{\alpha-1}(u-x)^{\beta-1}}{(u-l)^{\alpha+\beta-1}\int_0^1 t^{\alpha-1}(1-t)^{\beta-1}dt} \qquad l \le x \le u; \alpha, \beta > 0, \tag{4.4}$$

where α and β are shape parameters [45].

Figure 4.2 plots the shape of four beta probability distribution curves resulting from four different pairs of shape parameters, specifically $\alpha = 2$, $\beta = 4$; $\alpha = 3$, $\beta = 9$; $\alpha = 4$, $\beta = 16$; and $\alpha = 5$, $\beta = 25$. In DAGs with beta distribution task weights, the shape of the weight distribution for each of the tasks is selected randomly (with equal probability) from these four pairs of shape parameters.

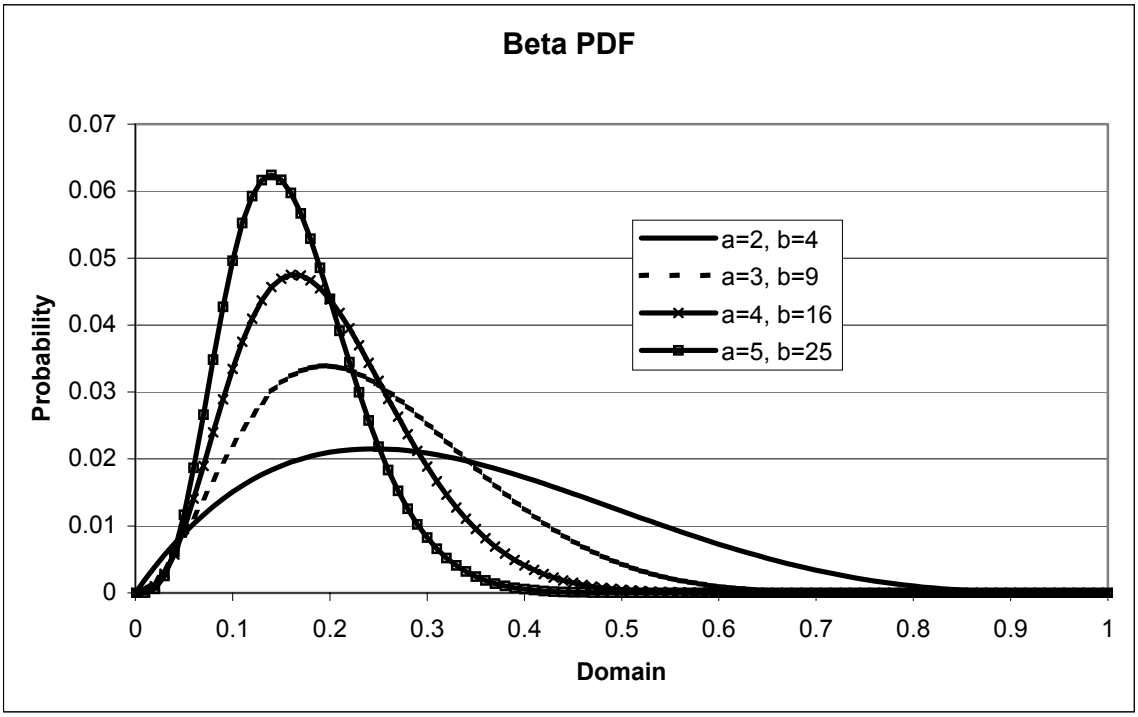


Figure 4.2 Beta Probability Distribution with a Variety of Shape Parameters

The second type of probability distribution used to generate task weight PDFs is the exponential distribution [45]. This distribution is commonly used to model task execution times in stochastic scheduling of periodic tasks [10, 61]. The exponential distribution is computed using the following equation:

$$\pi(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & otherwise \end{cases}, \tag{4.5}$$

where λ is the shape parameter and $\lambda > 0$. The shape of the exponential distribution with $\lambda = 1$, and $0 \leq x \leq 100$ is plotted in Figure 4.3.

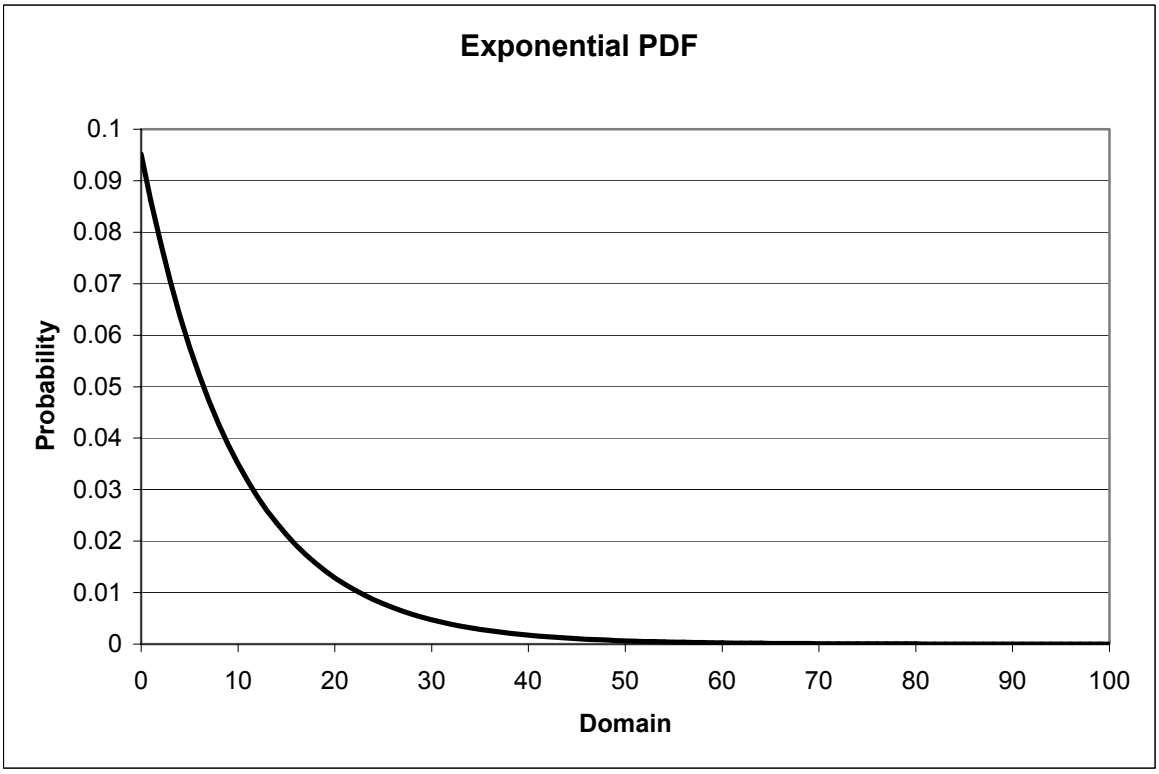


Figure 4.3 Exponential Probability Distribution with $\lambda = 1$

The third type of probability distribution used to generate task weight PDFs is the *randomized* distribution. This distribution is a variant of the uniform distribution [45]. However, unlike the smooth line of the uniform distribution, the randomized distribution closely models the peaks and valleys found in realistic PDFs (*e.g.*, in figures 3.3 and 3.4). This distribution is generated by selecting a random number from a uniform distribution

for each point in the weight PDF and dividing each point in the weight PDF with the sum of the selected random numbers. Figure 4.4 plots an example of the randomized probability distribution.

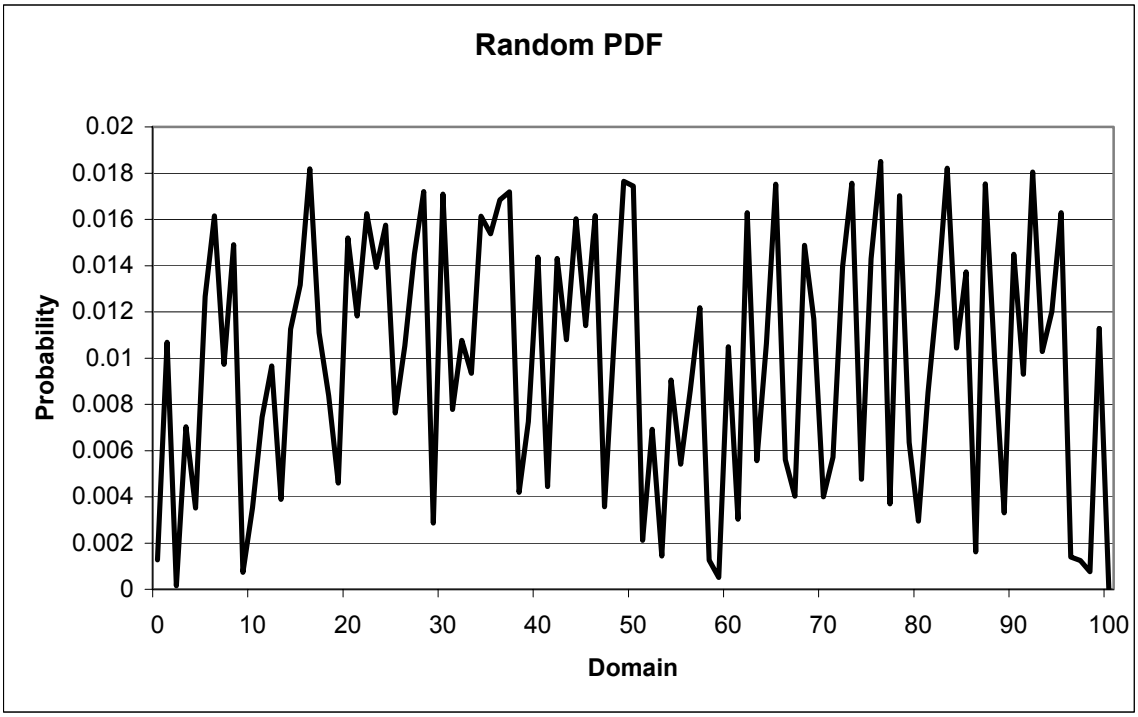


Figure 4.4 Randomized Probability Distribution

4.1.4 DAG sizes

For each of the HFJ, MVA, OUT, Random, and SFJ DAG structures, several DAGs with a total number of tasks (*i.e.*, sum of vertices and edges) in the ranges [290, 325], [390, 425], and [490, 525] were created. The FFT structured DAGS have exactly 605 tasks (*i.e.*, 223 vertices and 382 edges) resulting from 32-way butterfly concurrency. These DAG sizes were selected in order to provide an additional degree of variability in

the DAGs and in order to investigate the effectiveness of the stochastic scheduling techniques for DAGs of different sizes.

4.2 Directed Acyclic Graph Instances

The combination of DAG structures, CCR options, probability distribution options, and size options results in a total of 240 DAGs as summarized in Table 4.1.

Table 4.1 DAG Structure Combinations

Structure	Number of CCR Options	Number of Weight Distribution Options	Number of Size Options	Total DAGs
FFT	5	3	1	15
HFJ	5	3	3	45
MVA	5	3	3	45
OUT	5	3	3	45
Random	5	3	3	45
SFJ	5	3	3	45
			Total DAGs:	240

4.3 Experimental Parameters

This section describes the parameters used to control the stochastic schedule construction experiments performed as part of this dissertation. The two fundamental experimental options are the problem posed by the DAG and stochastic scheduling techniques used to construct a schedule for the DAG. Schedules are constructed for each of the 240 DAGs using the three LS approaches using both the exact and estimate methods. For the exact method, separate schedules using nine slot-fitting threshold values of 100%, 95%, 90%, 85%, 80%, 75%, 70%, 65%, and 60% were constructed.

For the estimate method, separate schedules using 11 different task weight estimates were constructed. The task weights estimates were set to the weight at which the CDF of the task weight is 100%, 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, 10%, and 0%. Observe that the weight at which the task weight CDF is *x*% specifies the maximum execution time requirements for *x*% of the invocations of the task.

Both the exact method and estimate methods can also be used in the GLS stochastic scheduling approach. However, the strength of the GLS approach lies in constructing and evaluating thousands of different schedules for a given DAG. This requires the use of relatively fast schedule construction techniques in order to achieve good quality schedule quickly. Early results indicated that the computational costs of the exact method would result in a prohibitive increase in the schedule construction time as compared to the estimate method. Therefore, only the estimate method is used for computing start time and completion time PDFs in the GLS approach.

Furthermore, because of the relatively long time taken by the GLS to construct a schedule for a DAG, only the 15 FFT structured DAGS were used to construct schedules using GLS. The number of GLS experiments was further restricted by only using the WCET as the weight estimate for constructing the initial schedule before the tasks' start time and completion time PDFs are computed.

The various combinations of DAGs, scheduling techniques, and control parameters result in a total of 304,290 schedules as summarized in Table 4.2. The characteristics of each of these schedules are evaluated in order to establish the validity of

the hypothesis and to evaluate the effectiveness of the various scheduling parameters in controlling the quality of schedules produced.

Table 4.2 Summary of Scheduling Experiments

Heuristic	PDF Computation Method	No. of Slot-Fitting Threshold Values	Weight Estimate CDF Percent	No. of DAGs	No. of Schedules
SHLEFT	Exact	9	N/A	240	2,160
	Estimate	N/A	11	240	2,640
SETF	Exact	9	N/A	240	2,160
	Estimate	N/A	11	240	2,640
SCP	Exact	9	N/A	240	2,160
	Estimate	N/A	11	240	2,640
GLS	Estimate	N/A	1	90	90
				Total Number of Precursor Schedules:	14,490
				Number of Jitter Control Options per Precursor Schedule:	21
				Total Number of Schedules Constructed:	304,290

4.4 Metrics for Experiment Analysis

The primary focus of this research is to construct stochastic schedules for soft real-time systems. Therefore, the two most important characteristics of the schedules produced are schedule lengths and the probability that the schedule will meet end-to-end deadlines. A secondary objective is to study how well jitter can be controlled and what impact jitter control has on the probability of meeting deadlines. Another schedule characteristic that is of interest is how well resources are utilized. These metrics for schedule evaluation are presented in more detail in this section.

4.4.1 Stochastic Schedule Length

Stochastic schedule length, $M(x)$, is the minimum amount of time required to complete the schedule with a probability of x%. $M(x)$ is determined from the end-to-end completion PDF of the schedule. The end-to-end completion PDF is the maximum of the

completion PDFs of all the terminal vertices in the DAG. Let $V_T \subset V$ be the set of terminal vertices in the DAG and let f_{vi} be the completion time PDF of vertex $v_i \in V_T$. The end-to-end completion time PDF of the schedule is given by the following expression:

$$f_{schedule} = \max\{f_{vi} : v_i \in V_T\}. \tag{4.6}$$

The PDF $f_{schedule}$ is defined over the interval $[l_{fschedule}, u_{fschedule}]$, and therefore, $u_{fschedule}$ is the maximum schedule length (*i.e.*, when 100% probability of success is required). Similarly, any amount of time less than $l_{fschedule}$ will result in a 0% probability of meeting the deadline. In general, if deadlines must be met x% of the time, then the schedule length can be reduced and is given by the minimum point in time in the schedule when the CDF (*i.e.*, $F_{schedule}$) of the end-to-end completion time PDF has a value of x%. Therefore, the schedule length is given by the inverse end-to-end completion time CDF of the schedule at x%. Formally,

$$M(x) = F_{schedule}^{-1}(x);\ 0 \leq x \leq 1. \tag{4.7}$$

4.4.2 Schedule Compression

The *schedule compression* metric is the relative reduction in the width of the completion time PDF of the schedule that occurs when less than 100% probability of meeting end-to-end deadlines is acceptable. Consider the end-to-end completion time PDF of a schedule $f_{schedule}$ bounded by the interval $[l_{fschedule}, u_{fschedule}]$. Clearly, allocating less than $l_{fschedule}$ time units to the schedule will result in a 0% probability of meeting the deadline. Conversely, allocating more than $u_{fschedule}$ time units will essentially waste the

time units beyond $u_{fschedule}$. Therefore, the range of time by which the schedule length can be usefully reduced is $[l_{fschedule}, u_{fschedule}]$. Let $M(x)$ be the length of the schedule that results when the required probability of meeting end-to-end deadlines is x%. The schedule compression metric is computed as follows:

$$\zeta(x) = \frac{M(1.0) - M(x) + 1}{M(1.0) - M(0.0) + 1} = \frac{u_{fschedule} - M(x) + 1}{u_{fschedule} - l_{fschedule} + 1}, \tag{4.8}$$

where $0 \le x \le 1$. Note, however, that for this metric to be of value, x must be restricted to useful probabilities for meeting deadlines. In other words, while the maximum possible compression metric value of 100% is achievable, it requires close to 0% probability of meeting end-to-end deadlines be 0, which is absurd for practical real-time systems.

4.4.3 QoS-Performance Tradeoff

The *QoS-performance tradeoff* relates the reduction in required probability of meeting end-to-end deadlines to the resulting schedule compression. The QoS-performance tradeoff metric of a schedule is computed as follows:

$$\xi(x) = \log_e \left(\frac{\frac{M(1.0) - M(x) + 1}{M(1.0) - M(0.0) + 1}}{1.0 - x} \right); 0 \le x < 1. \tag{4.9}$$

The QoS-performance tradeoff metric is computed on a logarithmic scale because a small reduction in required end-to-end probability can result in relatively large reductions in schedule as is evident in Chapter V. Scaling the metric values on the logarithmic scale makes the comparison between different metric values more intuitively meaningful.

4.4.4 Relative Schedule Length Improvement

The *relative schedule length improvement* metric provides a means for performing a comparative evaluation of the lengths of schedules produced by different scheduling heuristics and different scheduling control parameters. Relative schedule length improvement of two schedules is computed as follows. Let M_1 and M_2 be the maximum schedule lengths of two schedules $schedule_1$ and $schedule_2$, respectively. The relative schedule length improvement of the two schedules is computed as follows

$$\Psi(schedule_1, schedule_2) = \frac{M_1 - M_2}{M_1}. \tag{4.10}$$

Note that Ψ is negative when M_2 is greater than M_1 (*i.e.*, $schedule_2$ is worse than $schedule_1$).

4.4.5 Average Stochastic Jitter Factor

The *stochastic jitter factor* for a task in a schedule is the ratio of the jitter in the completion time of a task and the jitter in execution time requirements. The stochastic jitter factor for a task J_i with non zero weight is given by the following expression:

$$\Delta_i = \frac{u_{fi} - l_{fi} + 1}{u_{wi} - l_{fi} + 1}, \tag{4.11}$$

where $f_i(t)$ is the completion time PDF of task J_i defined in the interval $[l_{fi}, u_{fi}]$.

The *average stochastic jitter factor* is the average of the jitter factor over all tasks in the schedule. In computing the average stochastic jitter factor, tasks with zero weights (*e.g.*, edges that connect vertices scheduled for execution on the same processor) are not considered.

Let $E' \subset E$ be the set of edges in the DAG that have non-zero weights in the schedule and let V be the set of vertices in the DAG. The average stochastic jitter factor is computed as follows:

$$\Delta = \frac{\sum_{J \in V \bigcup E'} \frac{u_{fJ} - l_{fJ} + 1}{u_{wJ} - l_{wJ} + 1}}{|V| + |E'|}, \tag{4.12}$$

where the completion time PDF of task J is defined over the interval $[l_{fJ}, u_{fJ}]$ and the execution time requirement PDF of J is defined over the interval $[l_{wJ}, u_{wJ}]$.

4.4.6 Stochastic Footprint

The *stochastic footprint* is the sum of the count of the unit time slots in the schedule during which resources are reserved for execution by the schedule's tasks. This metric counts all time slots during which the slot may be used with any non-zero probability but does not include those time slots that are always idle (*i.e.*, gaps in the schedule).

Figure 4.5 depicts the resource utilization profile of a partial schedule of two tasks J_1 and J_2. Task J_1 is has a start time PDF of ⟨(11, 0.4), (12, 0.4), (13, 0.2)⟩ and a completion time PDF of ⟨(14, 0.2), (15, 0.2), (16, 0.2), (17, 0.2), (18, 0.2)⟩. Task J_2 is has a start time PDF of ⟨(17, 0.6), (18, 0.2), (19, 0.2)⟩ and a completion time PDF of ⟨(20, 0.1), (21 0.1), (22, 0.1), (23, 0.1), (24, 0.1), (25, 0.1), (26, 0.1), (27, 0.1), (28, 0.1), (29, 0.1) ⟩. The footprint of the two tasks is 19 time-units. Note that time units 11, 12, 15, 16, and 21 - 30, are all included in the footprint, even though they are used with less

than 100% probability. Conversely, time units 10 and 30 are not included in the footprint because they are not used by the schedule at all.

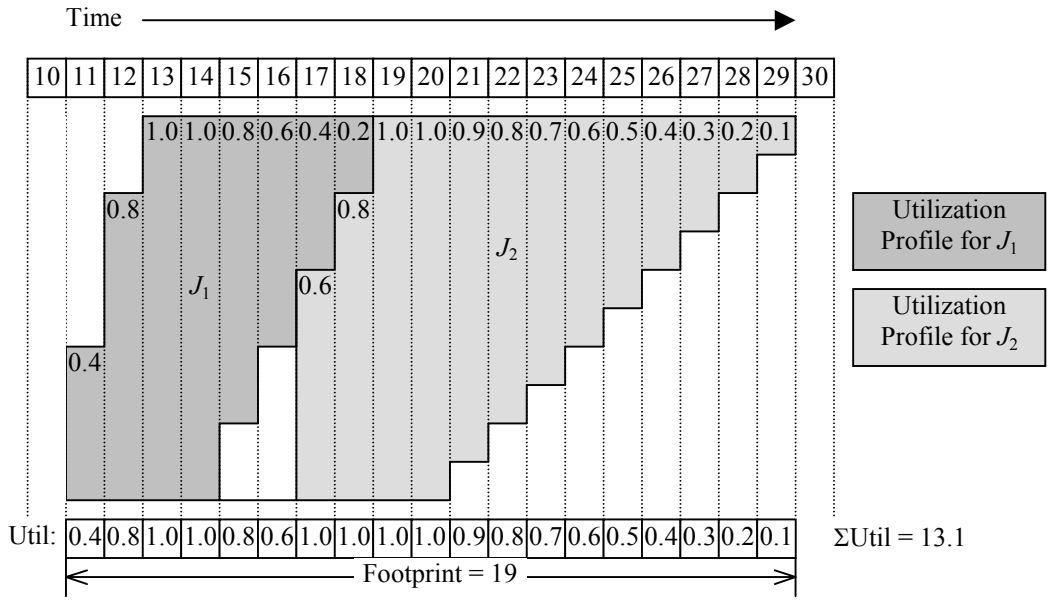


Figure 4.5 Profile of Resource Utilization of an Example Schedule with Two Tasks

It is significant that the stochastic footprint metric not include the count of 100% idle time slots in the schedule because the schedule, guarantees that these slots will not be used by the application and are available for use by other tasks (e.g., tasks of other applications, system support and maintenance tasks, etc.). The algorithm for computing the footprint of a stochastic schedule is given in Figure 4.6.

```
1.  footprint := 0
2.  start := 0
3.  let Ř:= set of resources in the schedule (i.e., all processors and send and receive
                                                                    links)
4.  Loop do ∀ř ∈ Ř:
5.      let ϑ:= set of tasks allocated to resource ř
6.      Loop do ∀J ∈ ϑ taken in increasing start time order
7.          let s_J be the start time PDF of J defined over the interval [l_sJ, u_sJ]
8.          let f_J be the end time PDF of J defined over the interval [l_fJ, u_fJ]
9.          if start < l_sj
10.             footprint := u_fJ – l_sJ + 1
11.             start := u_fJ
12.         else if start < u_fJ
13.             footprint := u_fJ – start
14.             start := u_fJ
```

Figure 4.6 Algorithm for Computing Stochastic Footprint

4.4.7 Stochastic Utilization

Stochastic utilization provides an insight into how effectively the stochastic schedule uses the available resources (*i.e.*, processors and communication links).

Let $E' \subset E$ be the set of edges in the DAG that have non-zero weights in the schedule and let V be the set of vertices in the DAG. Let $s_J(t)$ and $f_J(t)$ represent the start time and completion time of PDFs of task J, respectively. Stochastic utilization is given by the following expression:

$$\widetilde{U} = \frac{\sum_{J \in V \bigcup E'} \left[\sum_{l_{s,J}}^{u_{f,J}} \left(s_J(t) - f_J(t) \right) \right]}{footprint}, \tag{4.13}$$

where l_{sJ} is the lower bound of the interval over which $s_J(t)$ is defined, u_{fJ} is the upper bound of the interval over which $f_J(t)$ is defined, and *footprint* is the stochastic footprint

of the schedule. For the example in Figure 4.6, equation (4.13) evaluates to $13.1 \div 19 \approx 68.95\%$.

CHAPTER V

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents the results and analyses of the experiments conducted as part of this research. The first series of experiments evaluates the effects of using specific edge weight estimates on the stochastic schedule length and stochastic utilization when the estimate method is used to construct the stochastic schedules. The second series of experiments evaluates the effects of using specific slot-fitting thresholds when the exact method is used to construct the stochastic schedules. Next the estimate method is compared with the exact method for the three different LS heuristics in terms of stochastic schedule length.

Given the best combination of LS heuristic, PDF computation method, and scheduling control parameter, the ability to tradeoff probability of meeting end-to-end deadlines for schedule lengths is evaluated. Similarly, the ability to tradeoff probability of meeting deadlines for improved jitter is also studied for the best combination of scheduling parameters.

The final set of experiments evaluates the performance of the GLS approach. First, the stochastic schedule lengths of the schedules produced by the GLS are compared with those of the best schedules produced by the LS approaches. Next, the ability to tradeoff the probability of meeting end-to-end deadlines for schedule lengths in the

schedules produced by the GLS approach is evaluated. Similarly, the ability to tradeoff the probability of meeting deadlines for improved jitter is also analyzed for the GLS approach.

5.1 Stochastic List Scheduling Approach

This section presents and analyzes the performance of the three LS algorithms when the estimate method and the exact methods of computing task start time and completion time PDFs are used. The combination of three heuristics and two PDF computation methods results in a total of six LS algorithms. The six algorithms are designated as *Estimate SHLEFT*, *Estimate SETF*, *Estimate SCP*, *Exact SHLEFT*, *Exact SETF*, and *Exact SCP*.

5.1.1 Estimate Method

Figures 5.1 - 5.3 depict the improvement (or degradations – shown as negative improvement) in the maximum schedule length relative to the length of the WCET schedule that results when a variety of task weight scaling probabilities are specified to the estimate LS algorithms. These charts plot the average change in maximum schedule length grouped by DAG structure type, broken out by the three estimate algorithms. The *All* curve in the figures shows the averages across all the DAG structures. These figures show that the DAGs with different structure types respond differently to the scaling down of weights (relative to the WCET values) for the different algorithms. Figure 5.4 plots the improvement in maximum schedule length averaged across all estimate algorithms,

grouped by DAG structure type. This figure shows that, on average, scaling task weight estimate has little impact on schedule lengths when using the estimate LS algorithms.

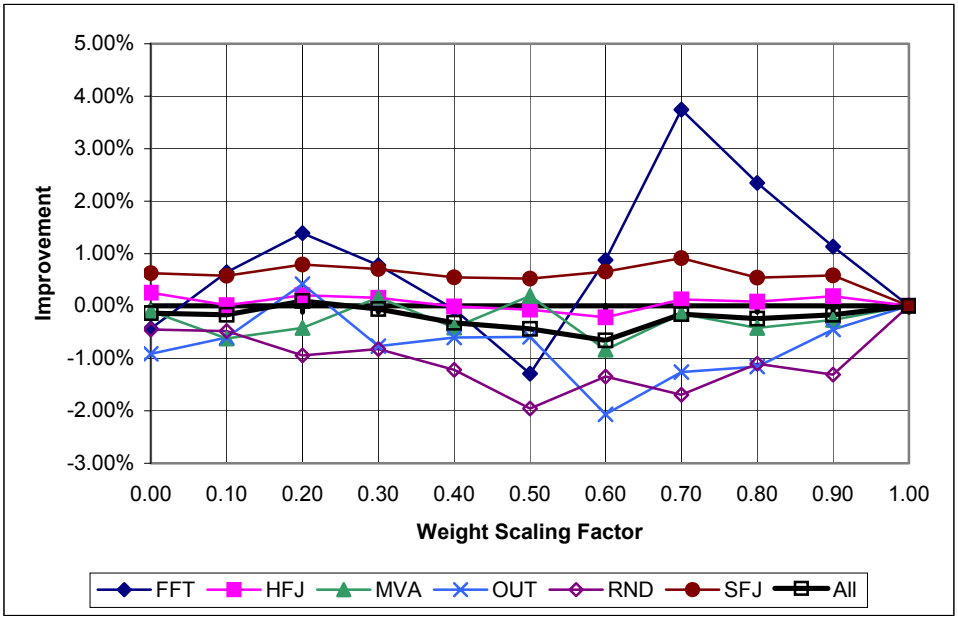


Figure 5.1 Schedule Length Improvement for Estimate SHLEFT Grouped by DAG Structure

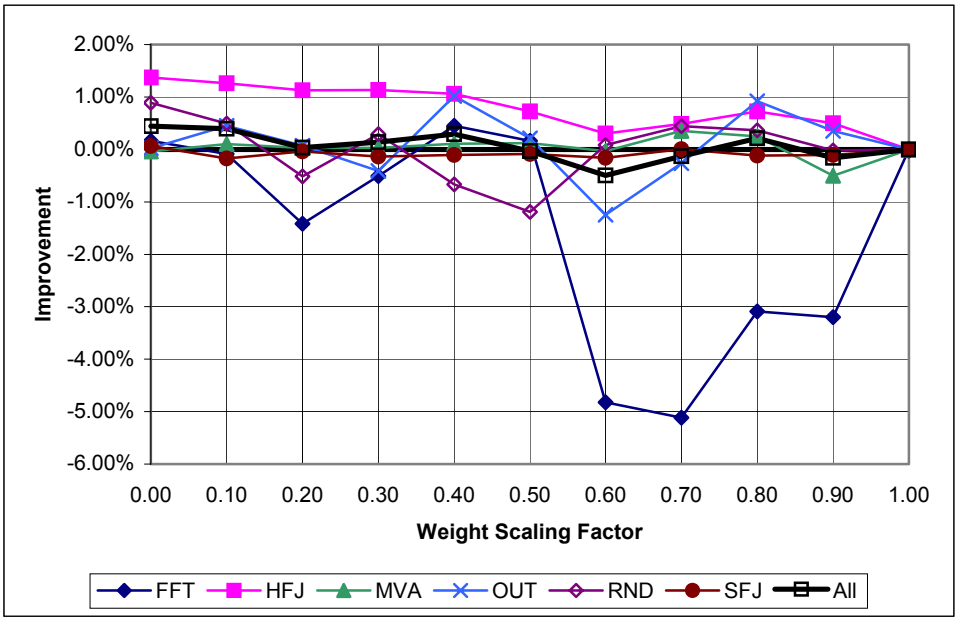


Figure 5.2 Schedule Length Improvement for Estimate SETF Grouped by DAG Structure

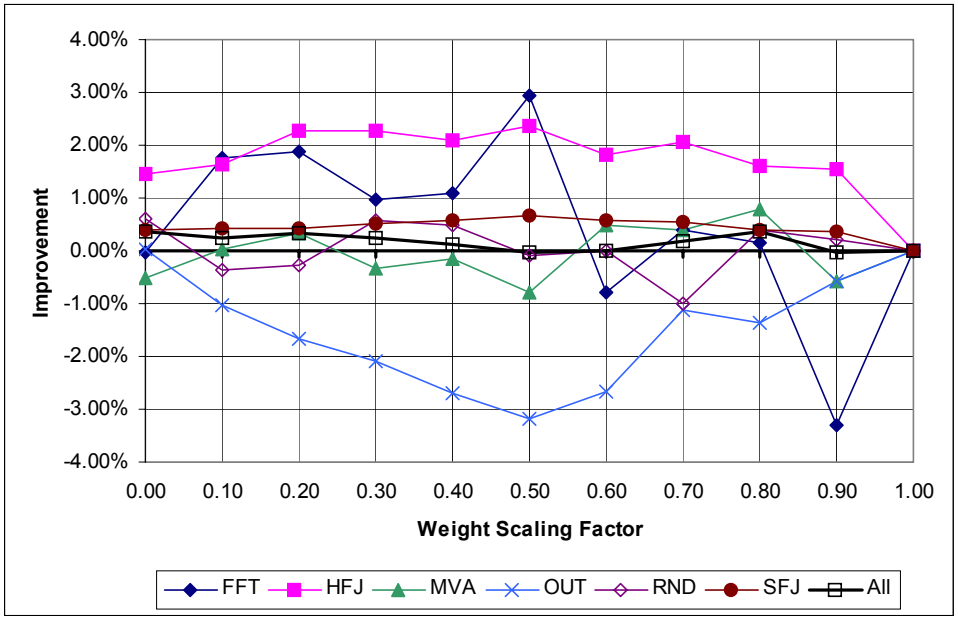


Figure 5.3 Schedule Length Improvement for Estimate SCP Grouped by DAG Structure

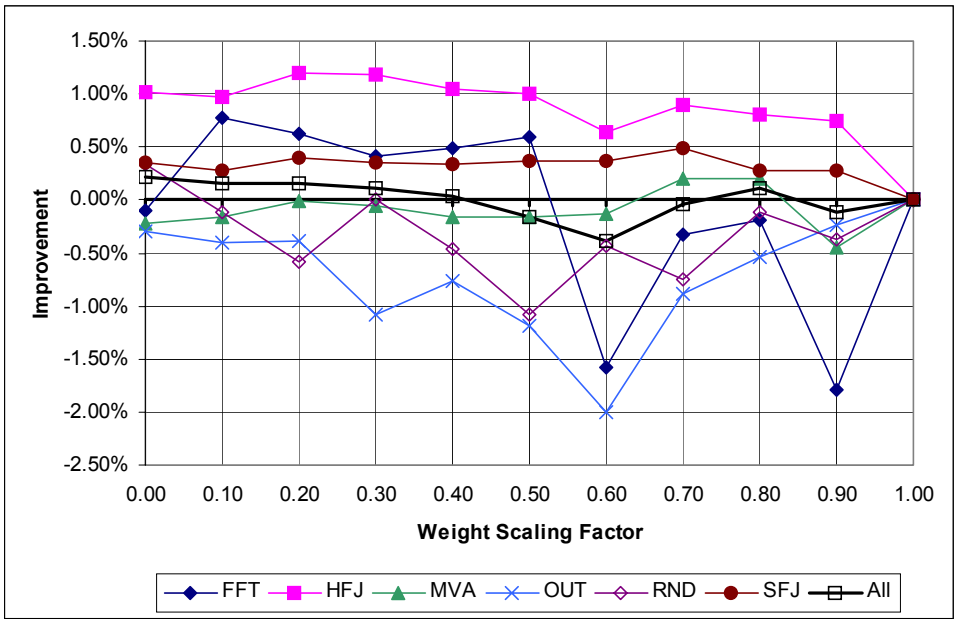


Figure 5.4 Schedule Length Improvement for All Estimate LS Algorithms Grouped by DAG Structure

Figures 5.5 - 5.7 plot the improvement in maximum schedule length grouped by weight probability distribution type, broken out by the three LS estimate algorithms. From this perspective also there is no significant trend in improvement or degradation of maximum schedule lengths. The improvement curves averaged over the three distribution types remain within a range of ±1.5%.

Figure 5.8 plots the improvement in maximum schedule length averaged across all estimate algorithms, grouped by weight distribution type. This graph shows that, in general, using a weight estimate that meets the execution time requirement of tasks approximately 60% of the time has the worst impact on schedule improvement (especially for DAGS with beta weight distributions). Furthermore, best-case execution

time produces a slightly better improvement (less than 1%) in schedule length compared with using WCET.

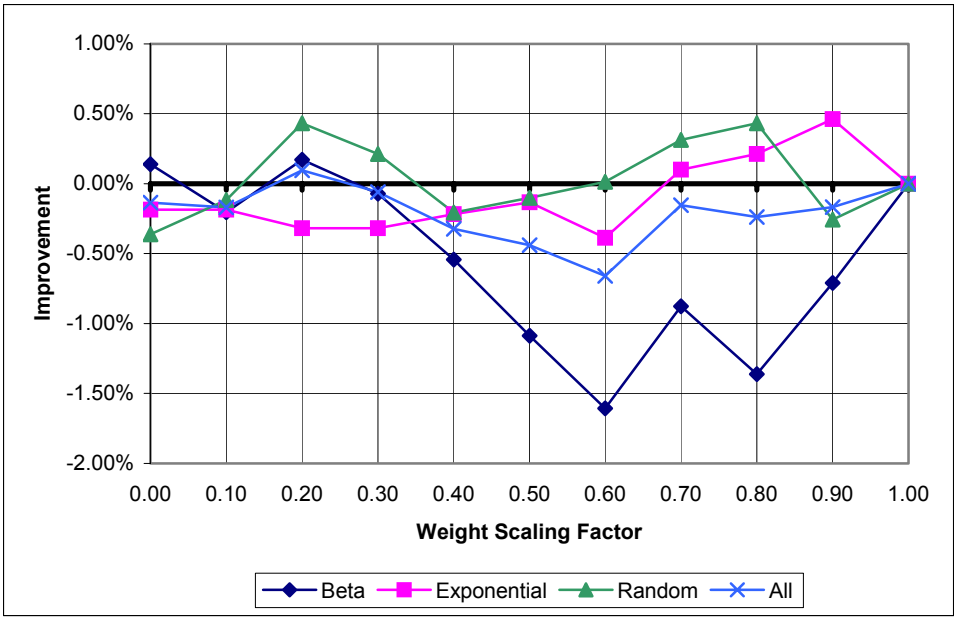


Figure 5.5 Schedule Length Improvement for Estimate SHLEFT Grouped by Weight Distribution

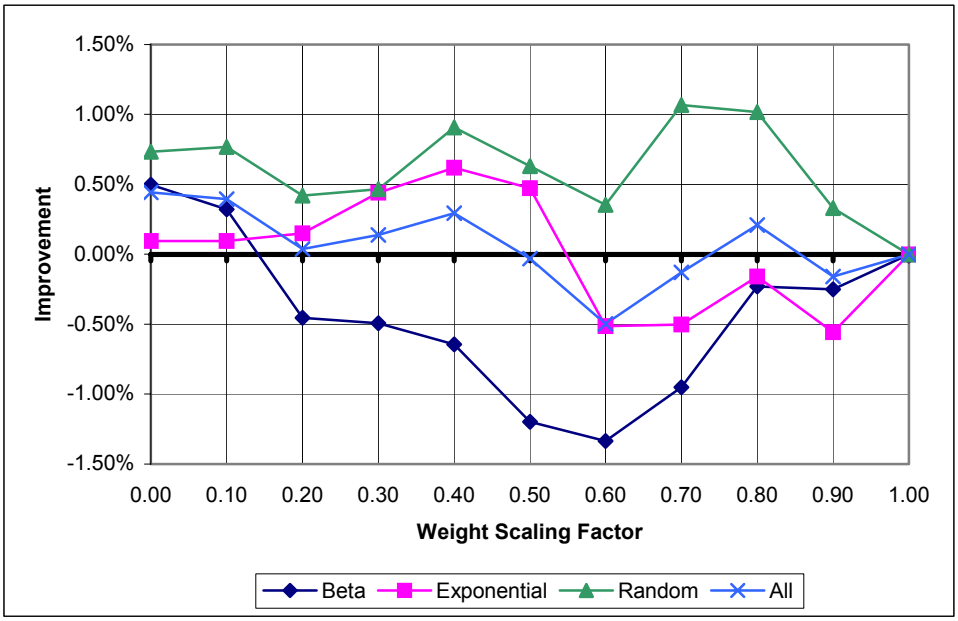


Figure 5.6 Schedule Length Improvement for Estimate SETF Grouped by Weight Distribution

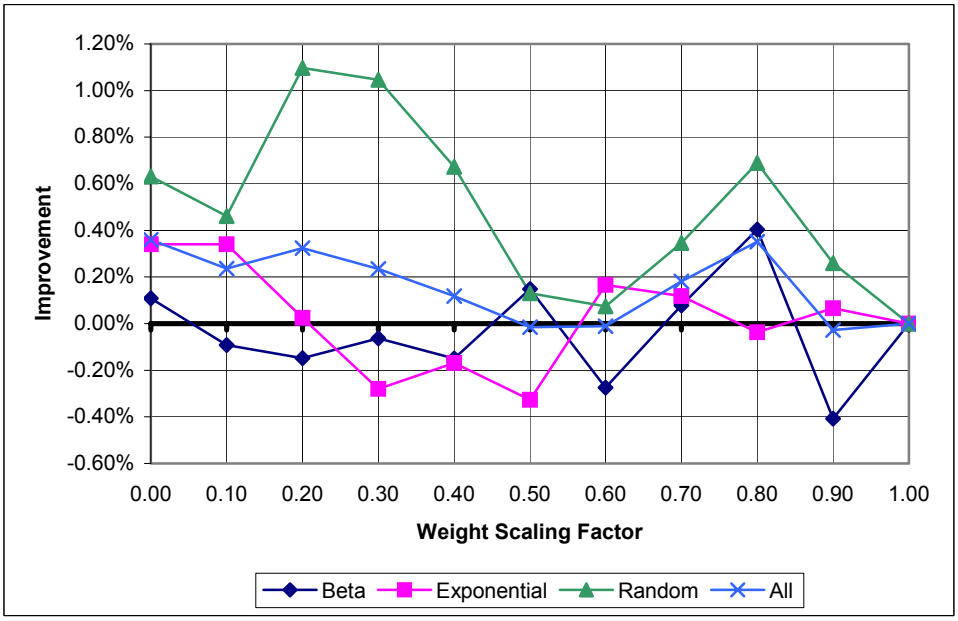


Figure 5.7 Schedule Length Improvement for Estimate SCP Grouped by Weight Distribution

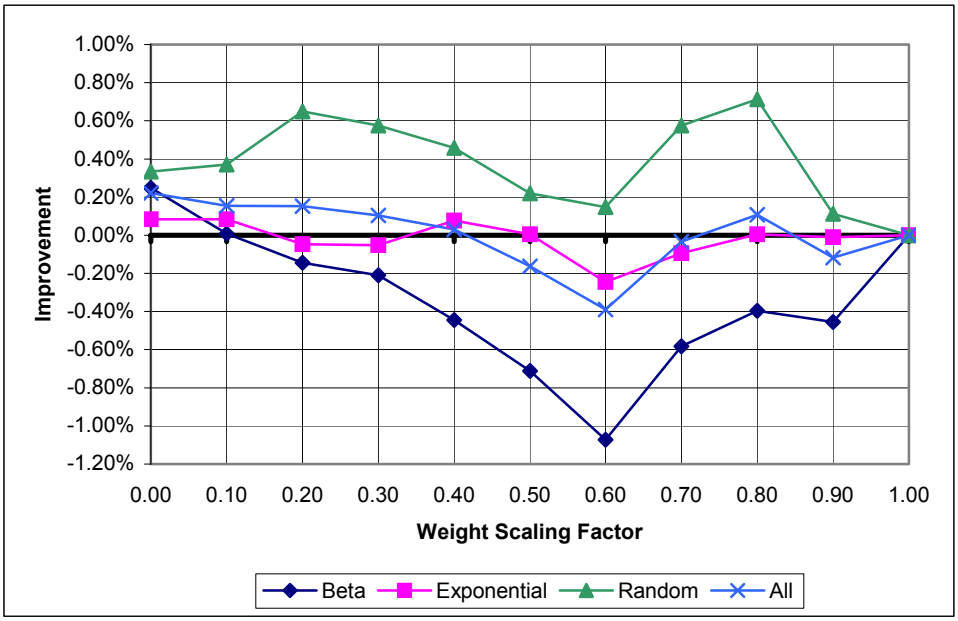


Figure 5.8 Schedule Length Improvement for All Estimate LS Algorithms Grouped by Weight Distribution

Figures 5.9 - 5.11 plot the improvement in maximum schedule length grouped by the DAGs' CCR, broken out by the three LS estimate algorithms. Figure 5.12 plots the improvement in schedule length averaged over all LS estimate algorithms, grouped by the DAGs' CCR. From these charts it is evident that the DAG's CCR has no significant impact on the lack of response of the estimate LS algorithms to varying the weight scaling probability parameter. The maximum schedule length improvement curves averaged over the three estimate algorithms remain within an interval of ±0.5%.

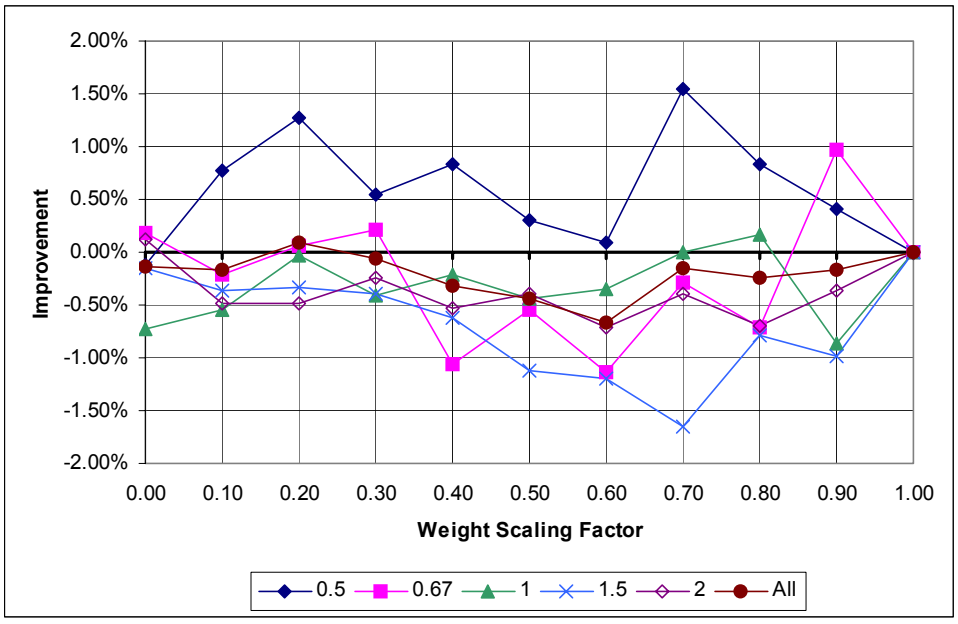


Figure 5.9 Schedule Length Improvement for Estimate SHLEFT Grouped by CCR

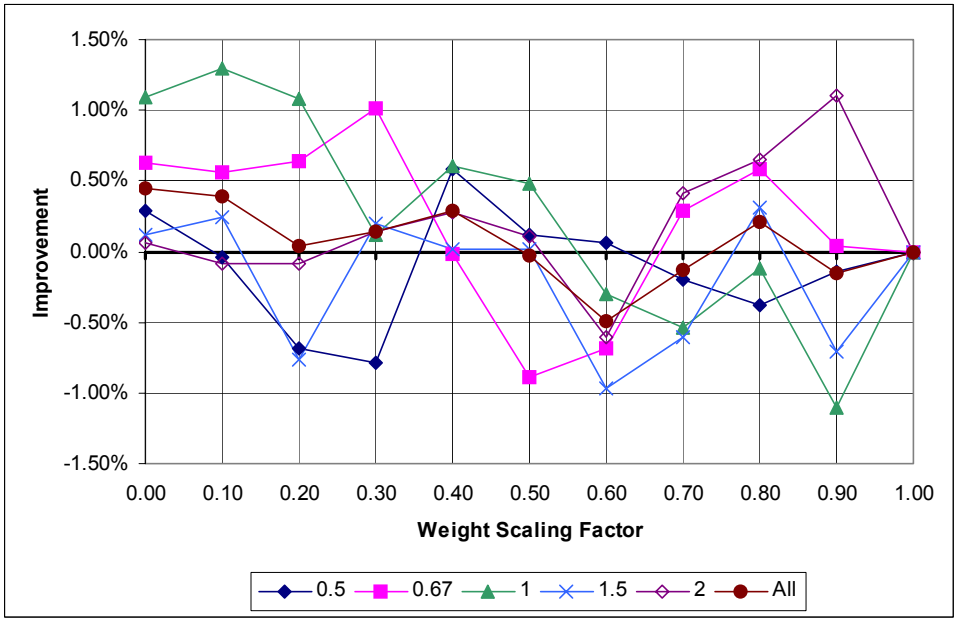


Figure 5.10 Schedule Length Improvement for Estimate SETF Grouped by CCR

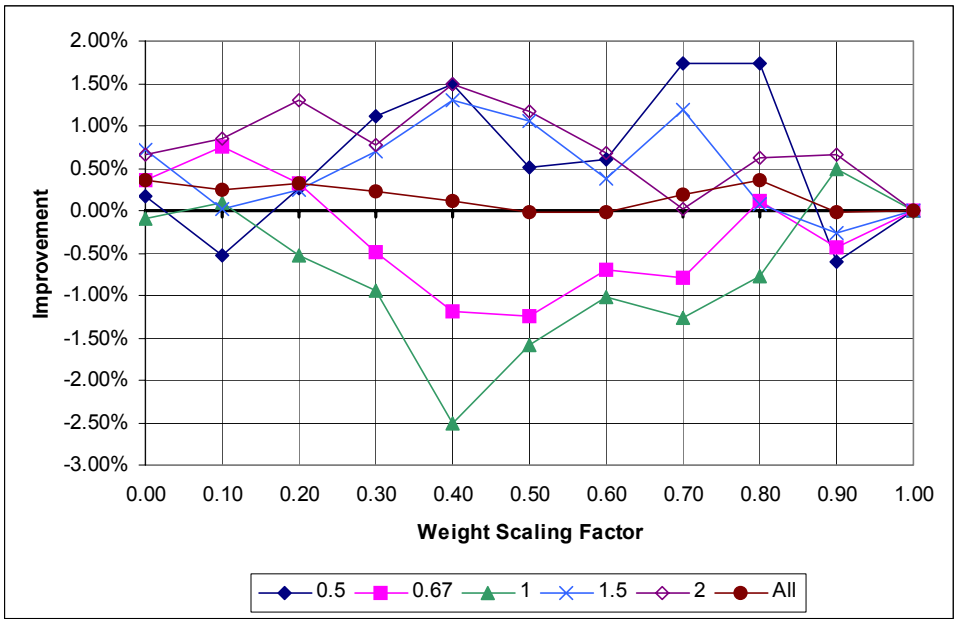


Figure 5.11 Schedule Length Improvement for Estimate SCP Grouped by CCR

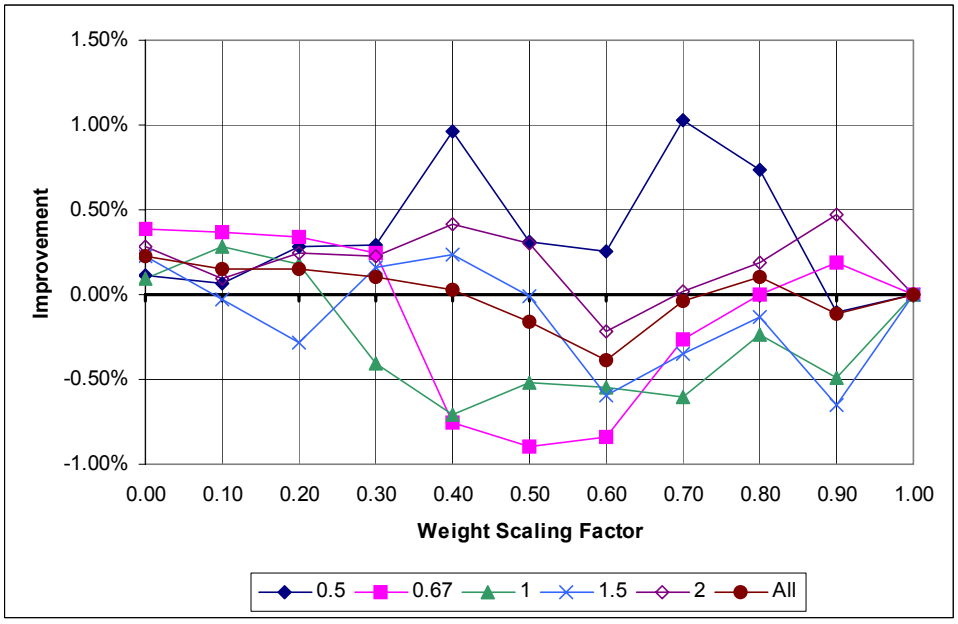


Figure 5.12 Schedule Length Improvement for All Estimate LS Algorithms Grouped by CCR

Figures 5.13 - 5.15 plot the improvement in maximum schedule length grouped by the DAGs' size, broken out by the three LS estimate algorithms. Figure 5.16 plots the improvement in schedule length averaged over all LS estimate algorithms, grouped by the DAGs' size. These charts also show that the DAG's size does not significantly improve or degrade the length of the schedules produced by the estimate algorithms when the weight scaling probability parameter is varied. The improvement curves averaged over the three estimate LS algorithms remain within an interval of ±0.4%. These graphs also show that, in general, using a weight estimate that meets the execution time requirement of tasks approximately 60% of the time has the worst impact on schedule improvement (this result is similar to what is seen in Figure 5.8).

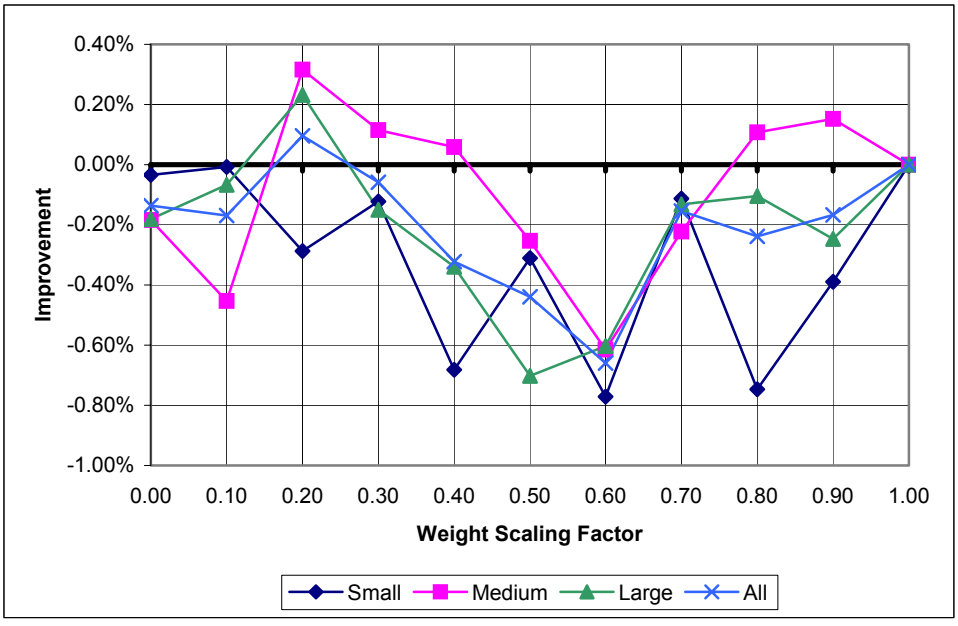


Figure 5.13 Schedule Length Improvement for Estimate SHLEFT Grouped by DAG Size

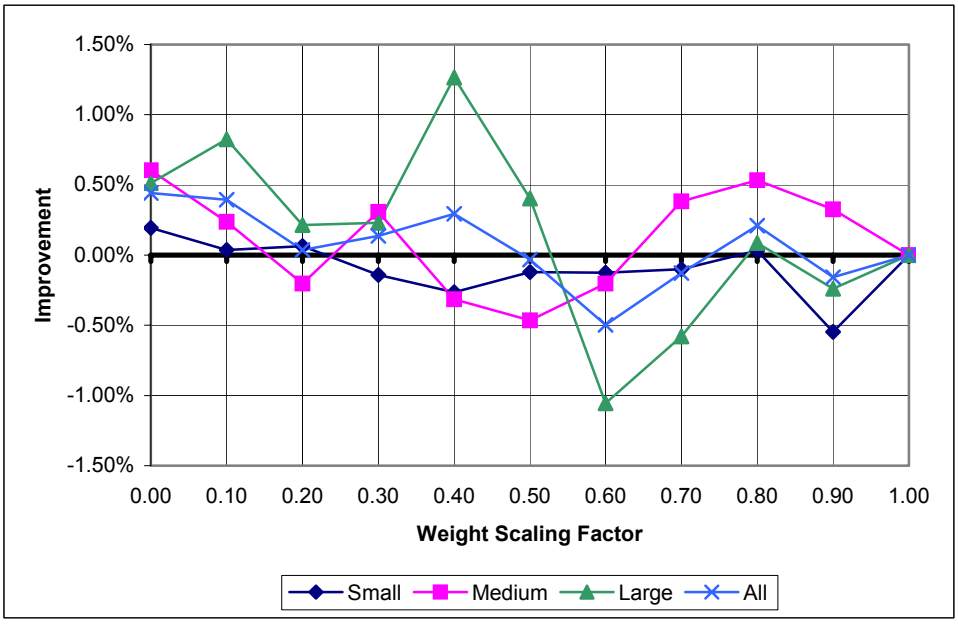


Figure 5.14 Schedule Length Improvement for the Estimate SETF Grouped by DAG Size

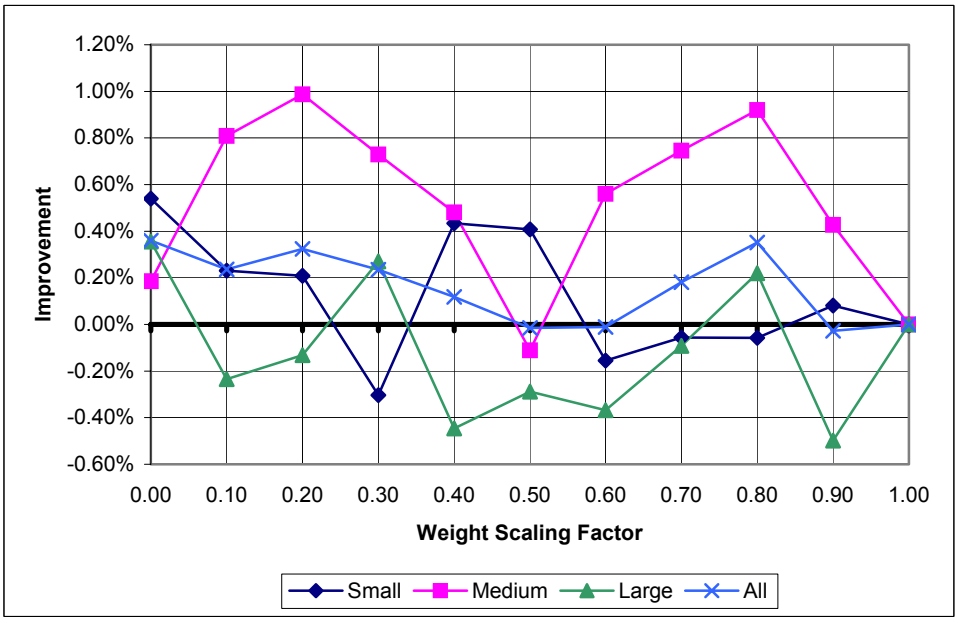


Figure 5.15 Schedule Length Improvement for Estimate SCP Grouped by DAG Size

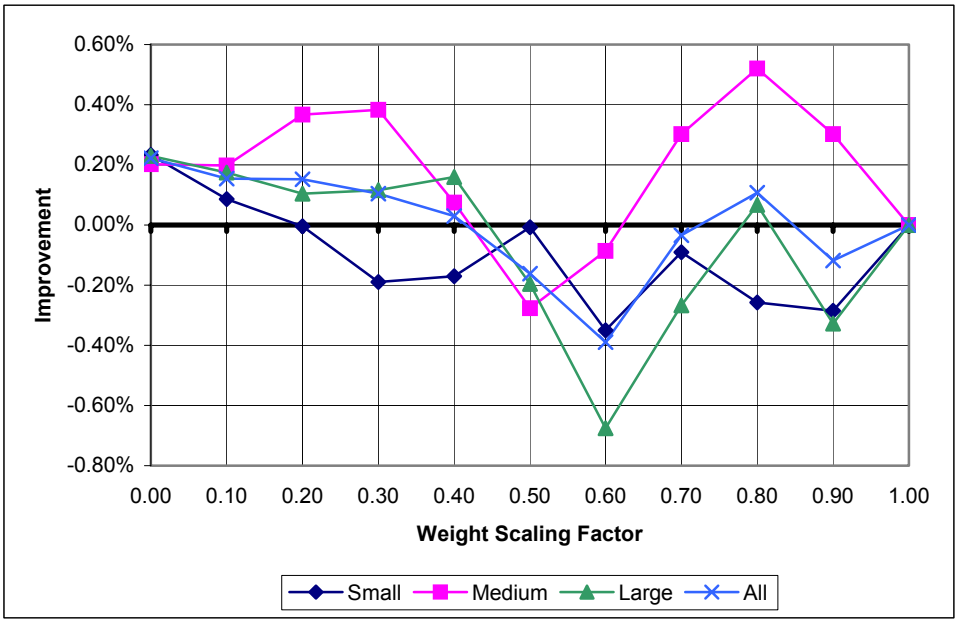


Figure 5.16 Schedule Length Improvement for All Estimate LS Algorithms Grouped by DAG Size

Figure 5.17 plots the improvement in maximum schedule length grouped by the three estimate LS algorithms taken over all DAGs. This chart provides clear evidence that the estimate algorithms will produce schedules with roughly the same length regardless of the how much tasks' weights are scaled down using the scaling probability parameter. However, using a weight estimate that meets the execution time requirement of tasks approximately 60% of the time will produce longer schedules than using WCET as an estimate. Although, on average, the schedule length is increased or decreased by less than 1% compared to using WCET, implying that the WCET, the best-case execution time, or the average weight can be used as the weight estimate to guide the first phase of the estimate scheduling algorithms. Note that the weight of all tasks in a DAG must be estimated uniformly (*i.e.*, using the same estimate scaling probability parameter). The

final schedule with accurate start and completion PDF of tasks is created in the second phase of the algorithm. This second phase ensures that sufficient time is allocated to each task in order to meet all possible execution time requirements even when the best case, average case, or other less than worst case weights are used as execution time requirement estimates during the first phase of estimate LS.

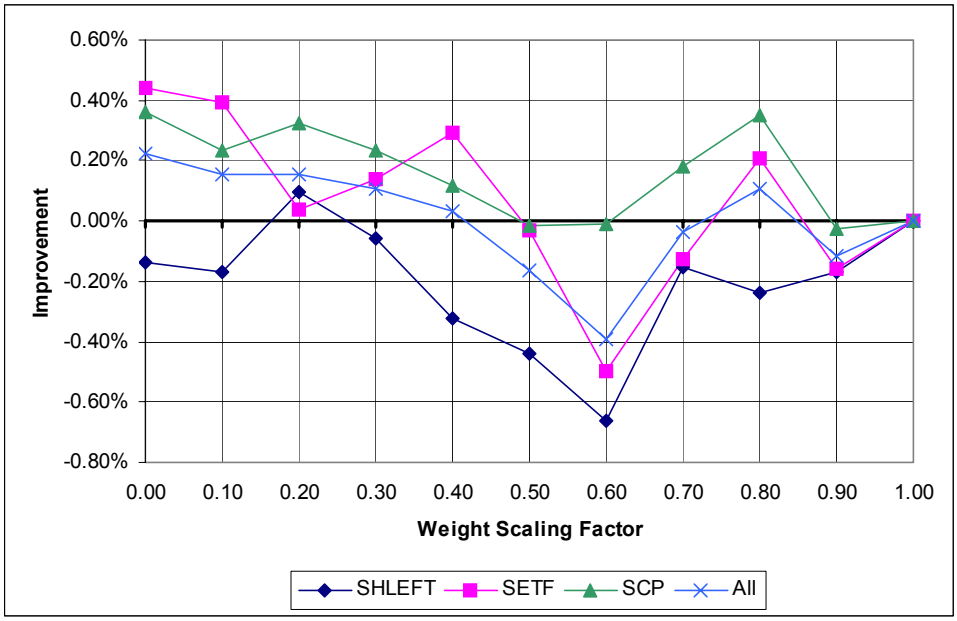


Figure 5.17 Average Schedule Length Improvement for All DAGs using the Estimate Methods

5.1.2 Exact Method

In the following series of charts, the improvement (or degradation – shown as negative improvement) in the maximum schedule length is depicted when a variety of slot-fitting threshold values are specified to the LS algorithms using the exact method for PDF computations. The improvement in maximum schedule length is relative to the

maximum schedule length that results when a slot-fitting threshold of 100% is specified. Recall that a threshold of 100% specifies that the task will not be inserted into an idle slot whose end time PDF bounding interval overlaps with the tasks' completion time PDF interval (*i.e.*, there is a chance that the task being considered for insertion will extend beyond the end of the idle slot). A threshold of less than 100% will allow tasks to extend beyond the end of the idle slot. Note that the minimum threshold value used is 60% in order to ensure that the task is guaranteed to fit in the idle slot when the task's weight is at least as much as the task's expected weight.

Figures 5.18 - 5.20 plot the relative improvement in the maximum schedule length grouped by DAG structure type, broken out by the exact LS algorithms. Figure 5.21 plots the improvement in the maximum schedule length for all exact LS algorithms, grouped by DAG structure type. The *All* curve in these charts shows the averages across all the exact LS algorithms. From these charts it is clear that the use of slot-fitting threshold probabilities less than 100% has a significant positive impact on the maximum schedule length for a variety of DAG-structure and LS heuristic combinations. Random structured DAGs, in particular, receive the most benefit from using a slot-fitting threshold of 95% or less. FFT and OUT DAGs scheduled using the exact SHLEFT algorithm are the only combinations that produced negative improvements. DAGs with the OUT structure scheduled using the exact SETF algorithm and DAGs with the SFJ structure scheduled using the exact SCP algorithm show no improvement or degradation (*i.e.*, have the same schedule length) regardless of the slot-fitting threshold value.

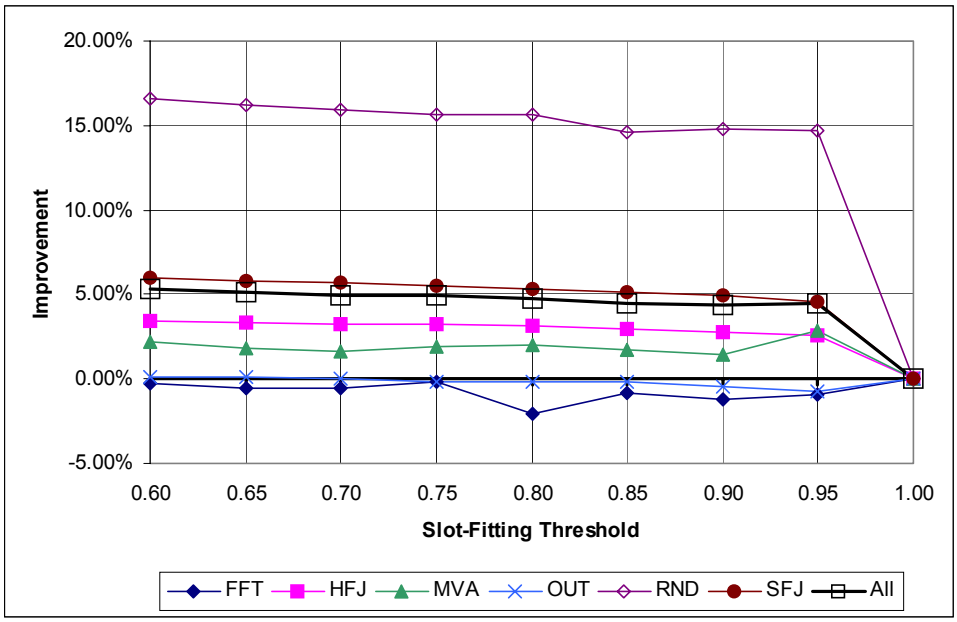


Figure 5.18 Schedule Length Improvement for Exact SHLEFT Grouped by DAG Structure

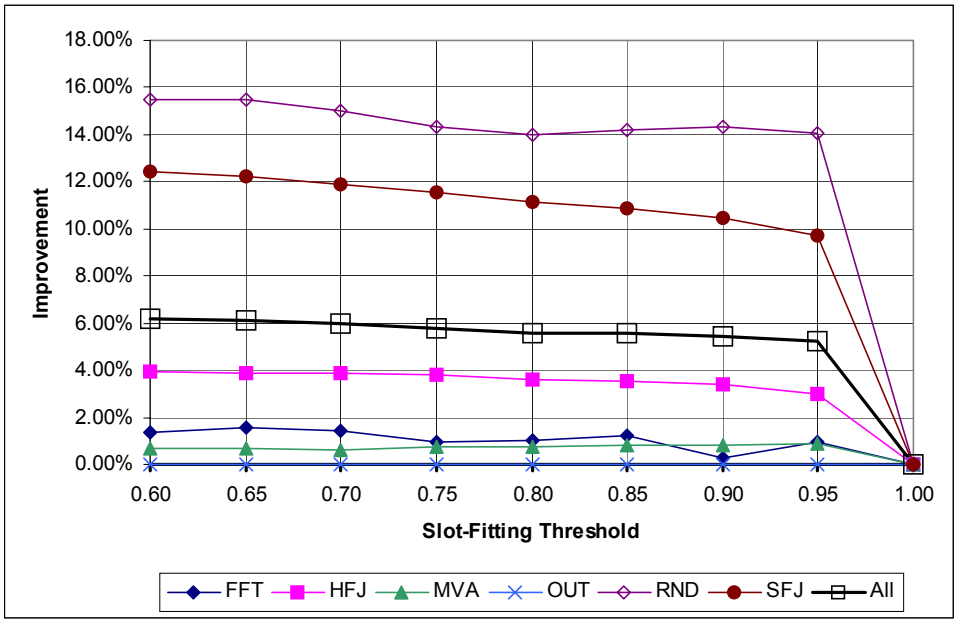


Figure 5.19 Schedule Length Improvement for Exact SETF Grouped by DAG Structure

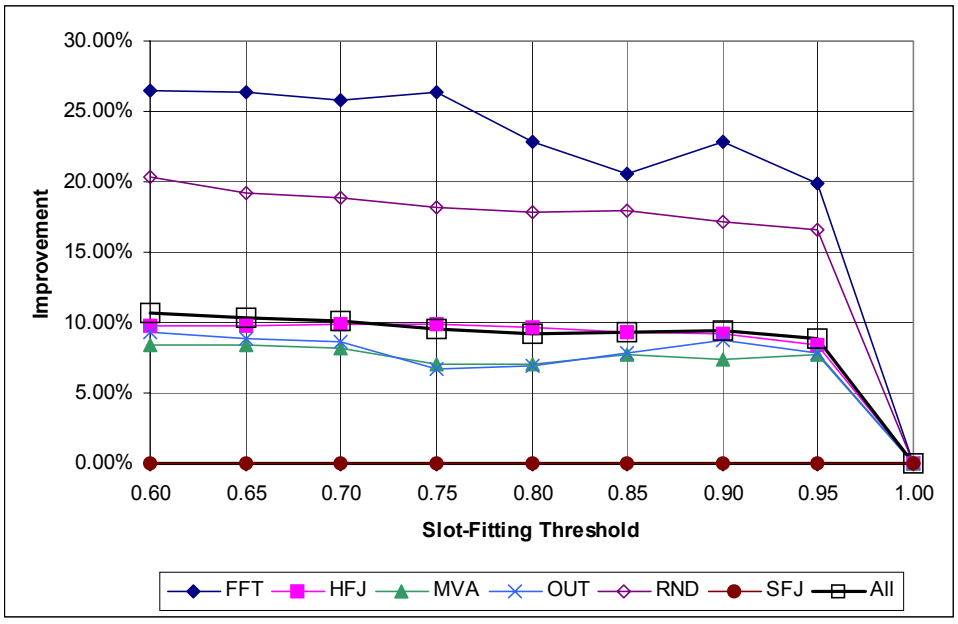


Figure 5.20 Schedule Length Improvement for Exact SCP Grouped by DAG Structure

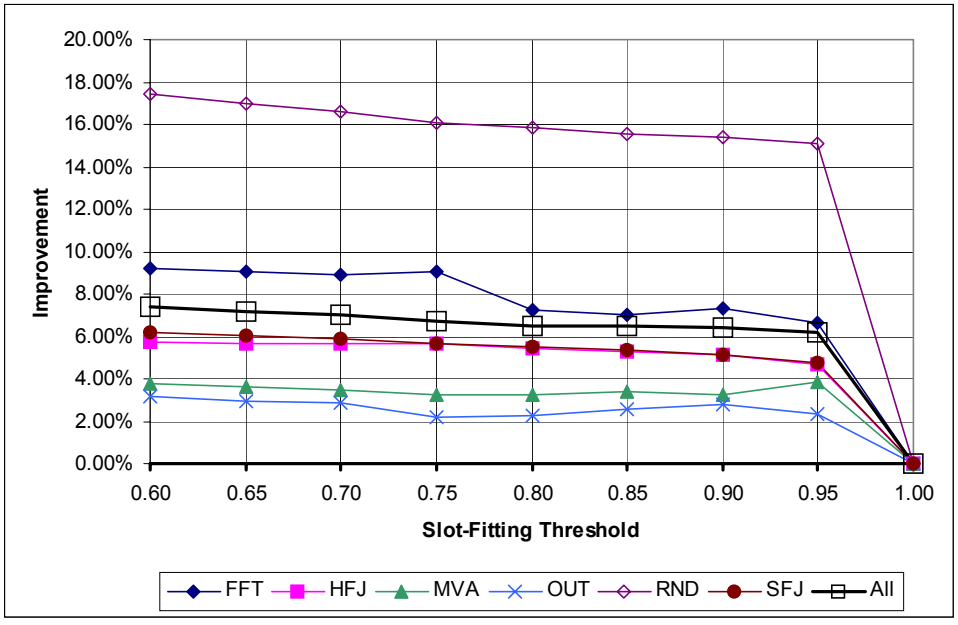


Figure 5.21 Schedule Length Improvement for All Exact LS Algorithms Grouped by DAG Structure

Figures 5.22 - 5.24 plot the relative improvement in the maximum schedule length grouped by task weight distribution type, broken out by the exact LS algorithms. Figure 5.25 plots the improvement in the maximum schedule length for all exact LS algorithms, grouped by task weight distribution type. In these charts, the maximum schedule length improvement averaged over all the exact LS algorithms is over 6% for DAGs with beta and exponential distributions, and over 9% for DAGs with random distributions. In general, DAGs with random task weight distributions derive the most benefit from reduced slot-fitting threshold values.

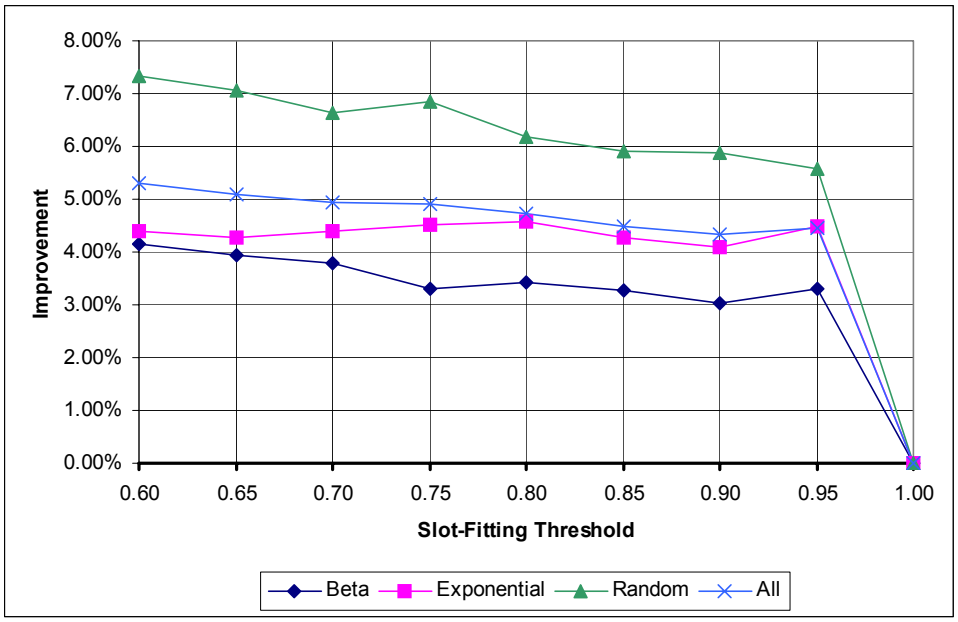


Figure 5.22 Schedule Length Improvement for Exact SHLEFT Grouped by Weight Distribution

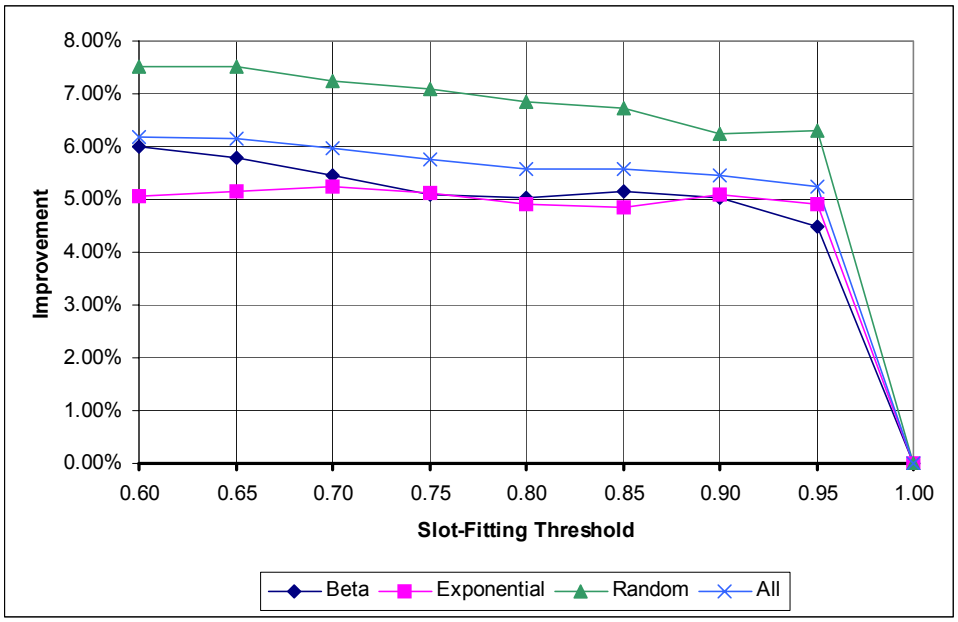


Figure 5.23 Schedule Length Improvement for Exact SETF Grouped by Weight Distribution

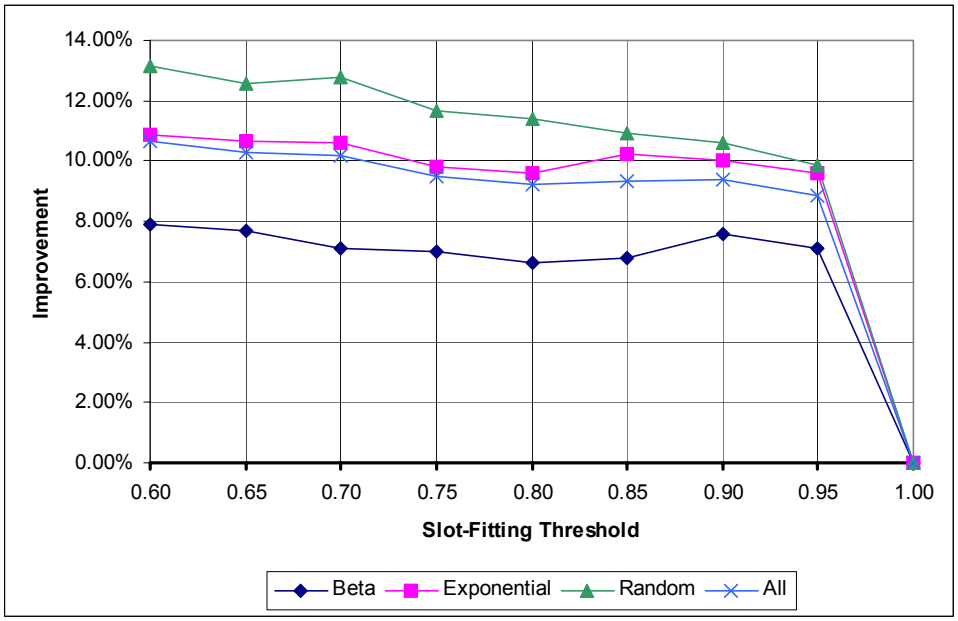


Figure 5.24 Schedule Length Improvement for Exact SCP Grouped by Weight Distribution

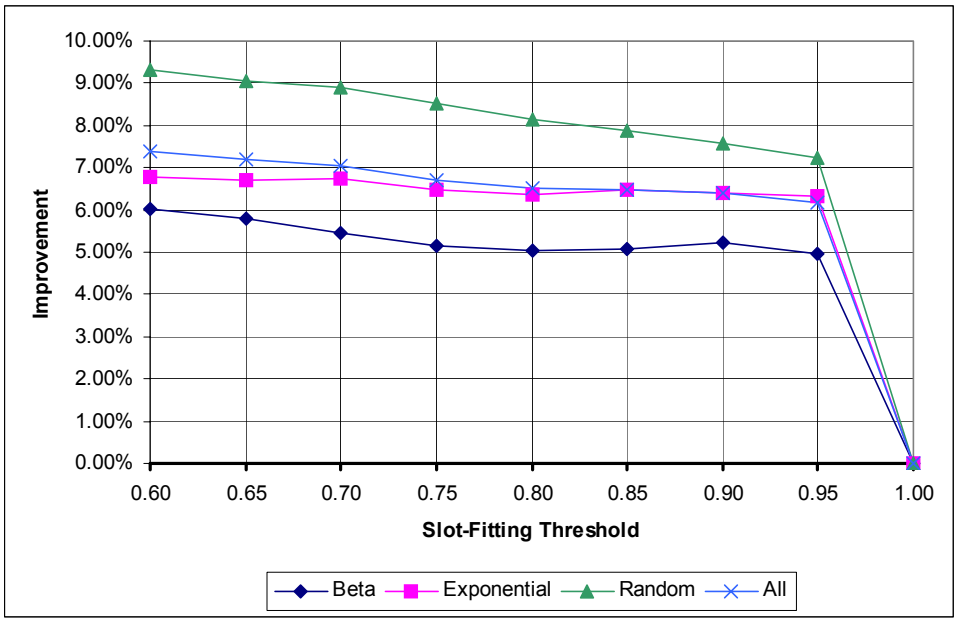


Figure 5.25 Schedule Length Improvement for All Exact LS Algorithms Grouped by Weight Distribution

Figures Figure 5.26 - Figure 5.28 plot the relative improvement in the maximum schedule length grouped by DAG CCR, broken out by the exact LS algorithms. Figure 5.29 plots the improvement in the maximum schedule length for all exact LS algorithms, grouped DAG CCR. In these charts, the exact SCP algorithm shows greater improvement in schedule lengths compared to the exact SHLEFT and exact SETF algorithms, in general. On average DAGs with CCR of 0.5 and 0.67 have below average maximum improvement in schedule length, DAGs with CCR of 1.0 have average maximum improvement, and DAGs with CCR of 1.5 and 2.0 have above average maximum improvement.

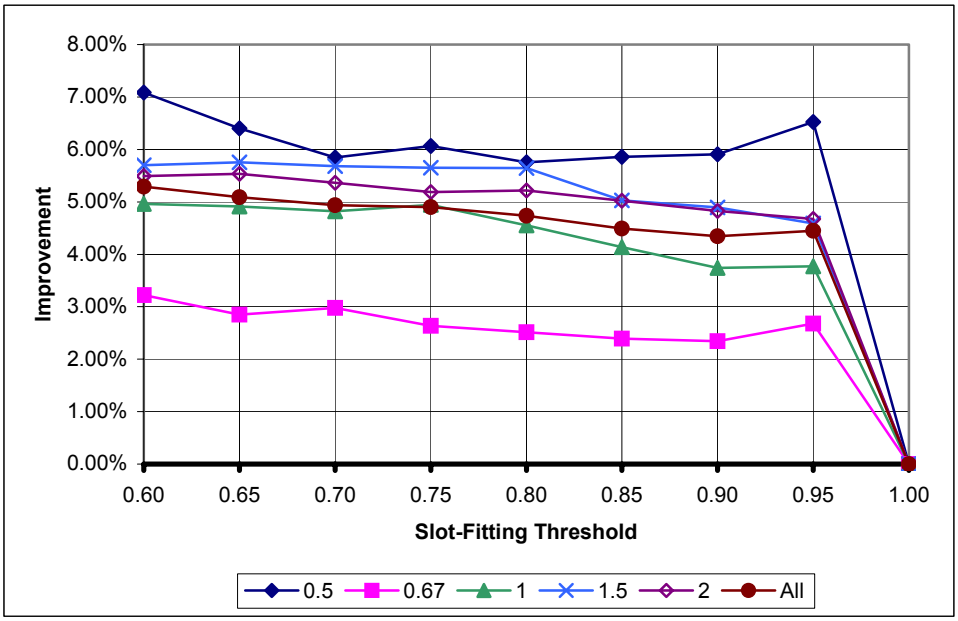


Figure 5.26 Schedule Length Improvement for Exact SHLEFT Grouped by CCR

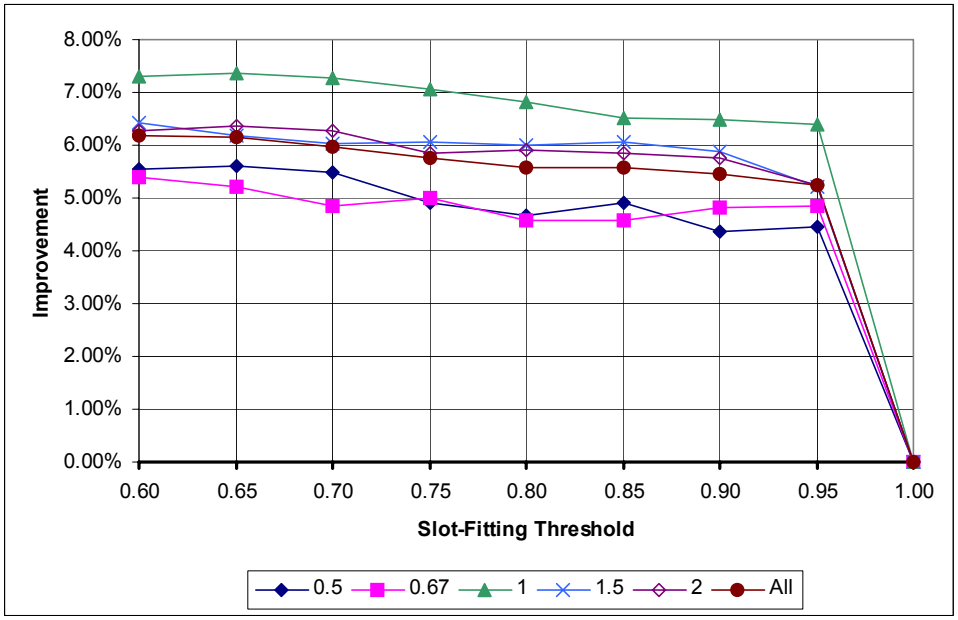


Figure 5.27 Schedule Length Improvement for Exact SETF Grouped by CCR

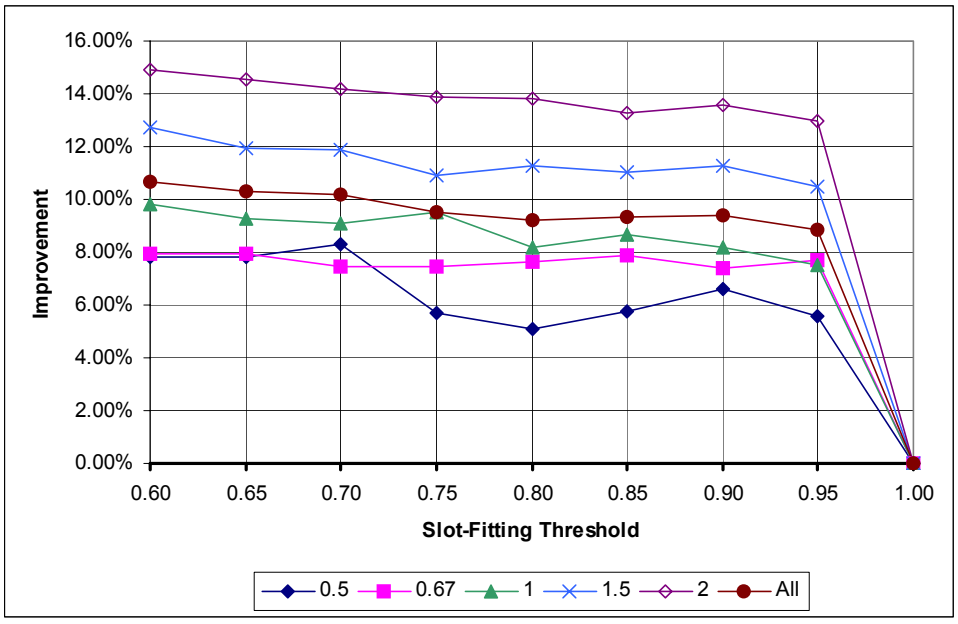


Figure 5.28 Schedule Length Improvement for Exact SCP Grouped by CCR

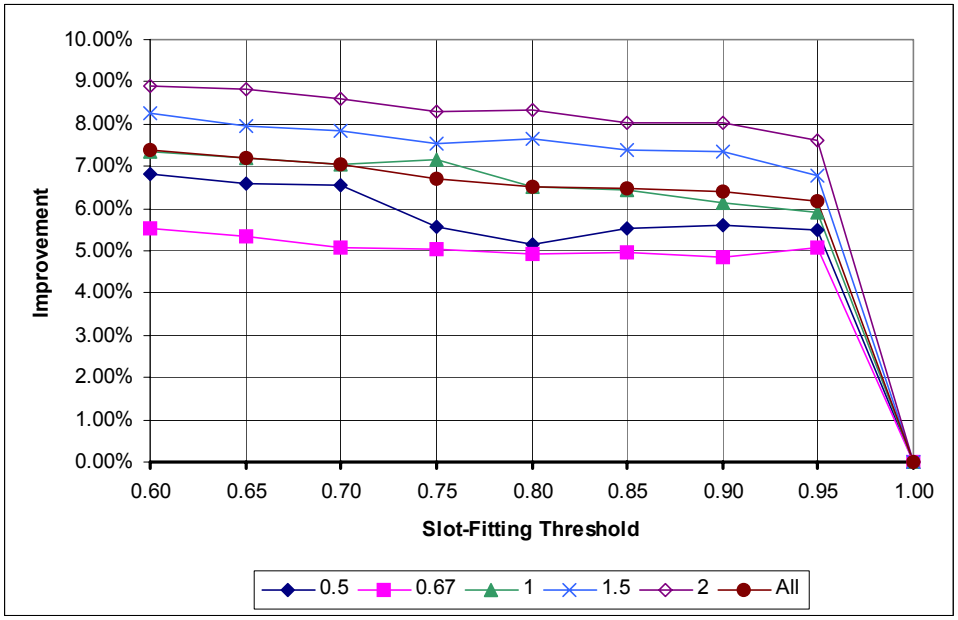


Figure 5.29 Schedule Length Improvement for All Exact LS Algorithms Grouped by CCR

Figures 5.30 - 5.32 plot the relative improvement in the maximum schedule length grouped by DAG size, broken out by the exact LS algorithms. Figure 5.33 plots the improvement in the maximum schedule length for all exact LS algorithms, grouped DAG size. These charts show that, in general, small, medium, and large DAGs have below average, approximately average, and above average maximum improvement, respectively. Furthermore, the maximum average improvement of the small, medium, and large DAGs is over 6%, over 7.5%, and over 8%, respectively.

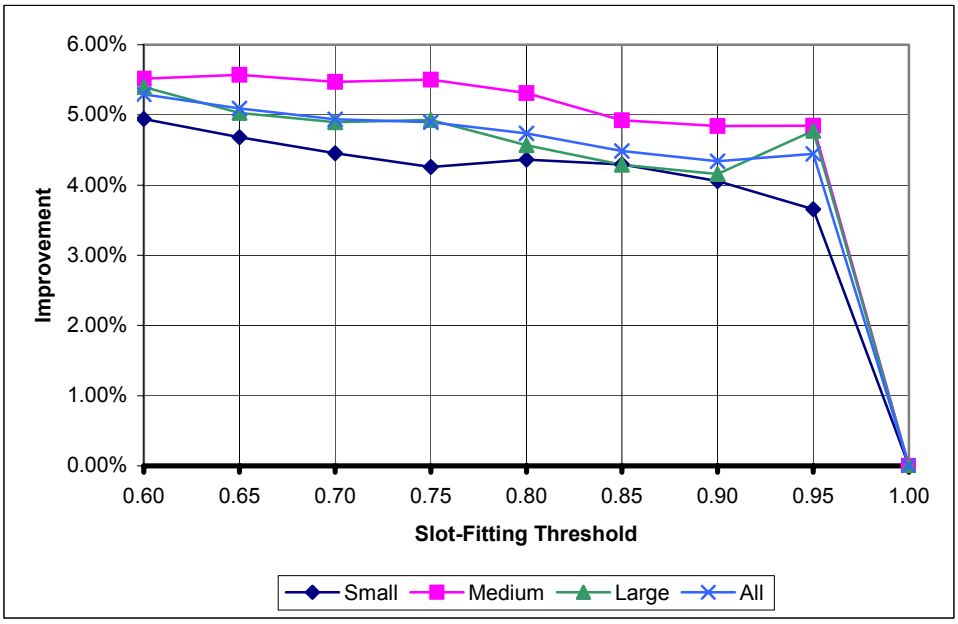


Figure 5.30 Schedule Length Improvement for Exact SHLEFT Grouped by DAG Size

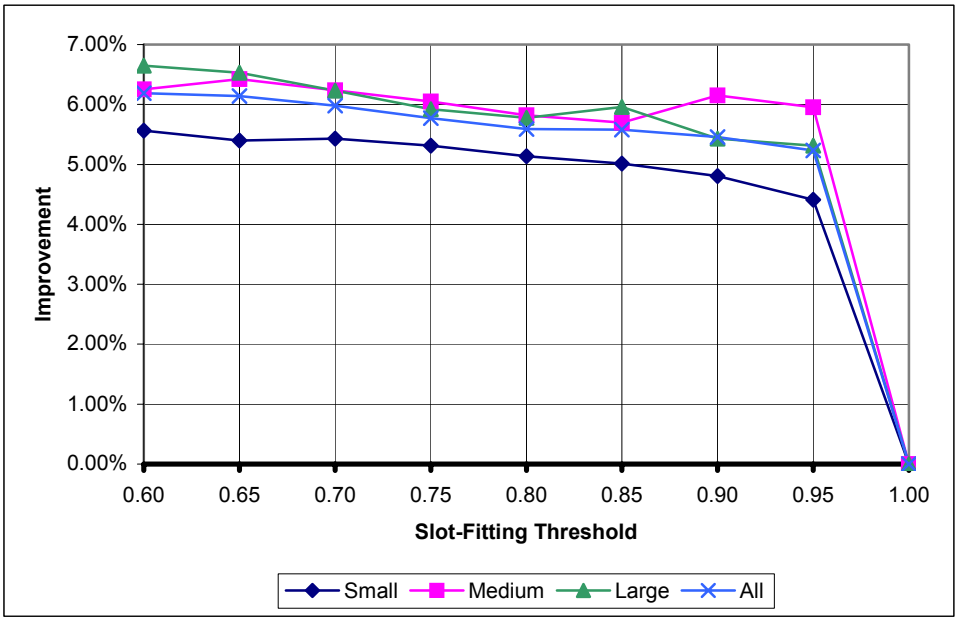


Figure 5.31 Schedule Length Improvement for Exact SETF Grouped by DAG Size

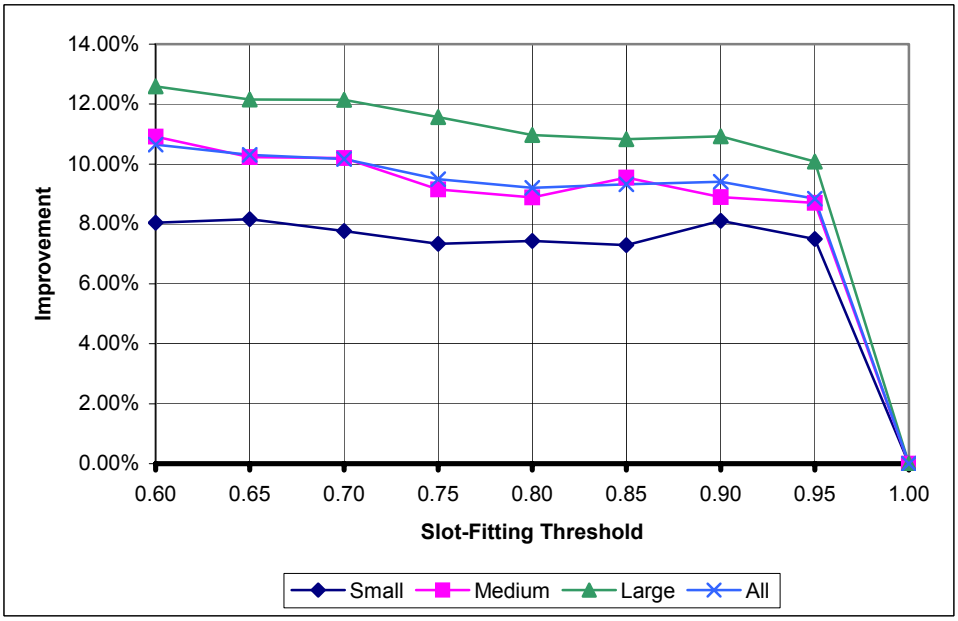


Figure 5.32 Schedule Length Improvement for Exact SCP Grouped by DAG size

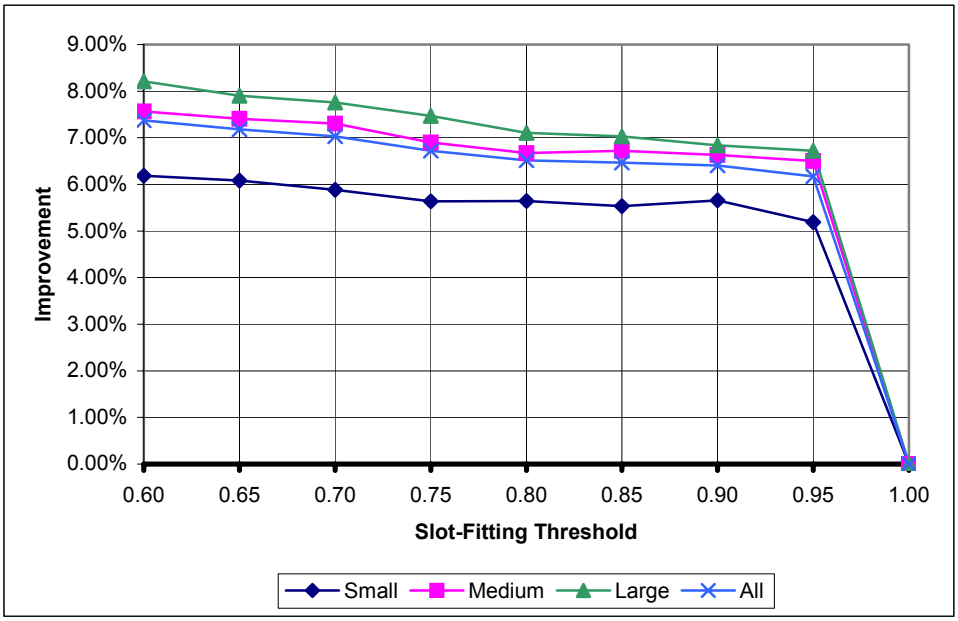


Figure 5.33 Schedule Length Improvement for All Exact LS Algorithms Grouped by DAG Sizes

Figure 5.34 depicts the relative improvement in the maximum schedule length grouped by the three exact LS algorithms taken over all DAGs. In this chart, the maximum schedule length improvement averaged over all the exact LS algorithms for all DAGs is over 7.0%. As expected, given the schedule length improvement charts above, the exact SCP algorithm shows significantly better improvement in schedule length as compared with the exact SETF and exact HLEFT algorithms.

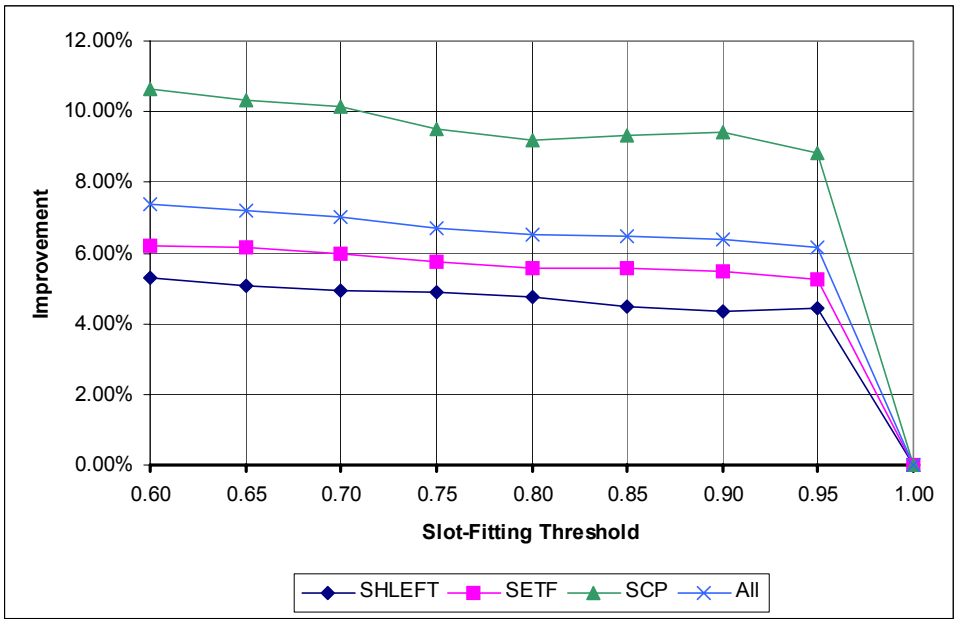


Figure 5.34 Schedule Length Improvement for All DAGs Using the Exact Method

5.1.3 Comparison of the Estimate LS and the Exact LS Methods

Recall that 11 different schedules (with the jitter control parameter fixed at 0.0) were produced for each of the 240 DAGs by each of the estimate-based LS algorithms using a variety of estimate parameters. Similarly, nine different schedules were produced for each DAG by each of the exact LS algorithms using a variety of slot-fitting threshold values. For each DAG-algorithm pair, the schedule with the least maximum schedule length was used as the basis of comparison between algorithms.

Table 5.1 summarizes the number of DAGs for which each LS algorithm produced the schedule with the shortest maximum schedule length when the probability of meeting the end-to-end deadline is 100%. The LS algorithms based on the exact PDF computation methods produce shorter schedules for 61.25% of the DAGs tested.

Furthermore, Table 5.2 shows that for the 54 DAGs for whom the exact SHLEFT algorithm produced the best schedules, the schedules were 47.97% shorter, on average, than the best schedule for the same DAGs using any of the estimate-based algorithms. Similarly, for 60 DAGs the exact SETF algorithm produced the shortest schedules that were 42.88% shorter, on average, than the best estimate-based schedules. And for 33 DAGs, the exact SCP algorithm produced the shortest schedules that were 37.49% shorter, on average, than the best estimate-based schedules.

The estimate LS algorithms produced the best schedules for 93 DAGs. For these DAGs, however, the improvement in schedule lengths of the estimate LS algorithms over the best exact LS algorithms is, on average, less than improvement in schedule lengths of the remaining 148 DAGs shown by the exact LS algorithms over the estimate LS algorithms.

Table 5.1 Comparison of LS algorithms

Algorithm	Best for Number of DAGs	Best for Percent of DAGs
Exact SHLEFT	54	22.50%
Exact SETF	60	25.00%
Exact SCP	33	13.75%
Estimate SHLEFT	10	4.17%
Estimate SETF	26	10.83%
Estimate SCP	57	23.75%

Table 5.2 Improvement of Schedule Lengths using Exact vs. Estimate LS

Best Overall Algorithm	Avg. % Schedule Length Improvement over Best Estimate Algorithm	Avg. % Schedule Length Improvement over Best Exact Algorithm
Exact SHLEFT	47.97%	N/A
Exact SETF	42.88%	N/A
Exact SCP	37.49%	N/A
Estimate SHLEFT	N/A	12.57%
Estimate SETF	N/A	21.60%
Estimate SCP	N/A	21.51%

Table 5.3 Comparison between Exact LS Algorithms

Algorithm	Best Exact Algorithm for Number of DAGs	Best Exact Algorithm for % of DAGs
Exact SHLEFT	70	29.17%
Exact SETF	83	34.58%
Exact SCP	87	36.25%

Table 5.4 Comparison between Estimate LS Algorithms

Algorithm	Best Estimate Algorithm for Number of DAGs	Best Estimate Algorithm for % of DAGs
Estimate SHLEFT	78	32.50%
Estimate SETF	44	18.33%
Estimate SCP	118	49.17%

Table 5.3 compares the performance of the three LS algorithms using exact PDF computations with each other. The exact SCP algorithm outperforms the other exact LS algorithms for 87 DAGs. However, because exact SETF outperforms the other exact LS algorithms for 84 DAGs, there is no clear best choice between the exact SCP and exact SETF LS algorithms. Table 5.4 compares the performance of the three estimate-based LS algorithms with each other. Here the SCP algorithm clearly outperforms the other

estimate-based algorithms for 118 DAGs. These results appear to indicate the SCP heuristic is superior to the others. However, because SCP does not outperform the other heuristics for a majority of the DAGs, the superiority of SCP cannot be established conclusively. Also note from Table 5.1 that SCP, SETF, and SHLEFT are best for 90, 86, and 64 DAGs, respectively. This indicates that SCP and ETF outperform each other in roughly equal number of cases, with SCP having a small advantage over ETF. It is clear that the simple SHLEFT heuristic does not perform as well as the other two heuristics.

Table 5.5 Ratio of Average Execution Times of Exact and Estimate LS

DAG Structure:	Size:	Execution Time Ratio of: Exact SHLEFT and Estimate SHLEFT	Exact SETF and Estimate SETF	Exact SCP and Estimate SCP
FFT	**Large**	18.19	295.06	28.27
HFJ	**Small**	17.76	66.96	22.01
	Medium	19.56	114.67	25.75
	Large	18.72	132.48	26.27
MVA	**Small**	12.10	49.05	15.53
	Medium	13.54	39.54	17.58
	Large	16.65	52.45	18.27
OUT	**Small**	19.64	475.78	23.74
	Medium	23.88	637.30	30.08
	Large	22.59	618.46	33.62
RND	**Small**	19.01	54.72	18.73
	Medium	23.23	71.70	25.10
	Large	28.09	105.74	29.25
SFJ	**Small**	17.58	55.08	15.83
	Medium	16.70	83.54	16.83
	Large	17.79	127.69	17.22
All	**Small**	17.04	97.24	18.05
	Medium	18.20	141.62	20.66
	Large	19.36	181.58	22.75

The execution times for the accuracy-based and estimate-based algorithms are compared for the various structures and sizes of DAGs in Table 5.5. The exact SHLEFT algorithm takes over 17 times, 18 times, and 19 times as much time as the estimate SHLEFT algorithm for the small, medium, and large sized DAGs, on average. Similarly, the exact SCP algorithm takes over 18 times, 20 times, and 22 times as much time as the estimate SCP algorithm for the small, medium, and large sized DAGs, on average. The exact ETF algorithm, on the other hand, takes over 97 times, 141 times, and 181 times as much time its estimate counterpart because SETF evaluates a much larger number of vertex-processor combinations that SHLEFT and SCP as explained below.

Let V_r be the set of ready vertices and P be the set of available processors on which ready vertices can scheduled. At each step of the LS algorithm, SHLEFT and SCP select the highest priority vertex and tentatively schedule it on each of the processors in P while looking for the processor that allows the selected ready vertex the earliest start time. After the best processor is found, the vertex is permanently scheduled. Therefore, at each step, SHLEFT and SETF compute $|P|+1$ schedules for the selected ready vertex.

On the other hand, SETF tentatively schedules every ready vertex on every processor in order to find the vertex-processor pair with the earliest start time, and then permanently schedules the earliest vertex-processor pair. Therefore, at each step, SETF computes a total of $|P|\times|V_r|+1$ schedules to allocate a single vertex. This accounts for the relatively large difference in schedule construction time for the exact SETF and estimate SETF algorithms.

Table 5.6 Relative Execution Times of the Exact LS Algorithms

		Relative Execution Times of Exact Method Algorithms		
DAG Structure	**Size:**	**SHLEFT**	**SETF**	**SCP**
FFT	Large	1.000	15.128	1.178
HFJ	Small	1.000	4.283	1.047
	Medium	1.000	5.719	1.052
	Large	1.000	6.748	1.041
MVA	Small	1.000	4.146	1.280
	Medium	1.000	4.175	1.388
	Large	1.000	4.297	1.296
OUT	Small	1.000	31.604	1.015
	Medium	1.003	41.954	1.000
	Large	1.000	37.353	1.067
RND	Small	1.000	3.763	1.013
	Medium	1.000	4.862	1.149
	Large	1.000	5.364	1.088
SFJ	Small	1.063	4.568	1.000
	Medium	1.058	6.140	1.000
	Large	1.038	8.880	1.000
All	Small	1.000	7.084	1.027
	Medium	1.000	9.598	1.050
	Large	1.000	10.949	1.075

Table 5.6 compares the execution times of the exact method LS algorithms relative to the exact method LS algorithm with the smallest execution time, broken out by DAG structure and size. The SHLEFT algorithm has the shortest execution time for most of the sample DAGs. The SCP algorithm has nearly the same execution time as the SHLEFT algorithm. As expected, the SETF algorithm has the longest execution time, especially for the out-tree DAG structure.

Table 5.7 compares the execution times of the estimate method LS algorithms relative to the estimate method LS algorithm with the smallest execution time, broken out by DAG structure and size. From this table, it is evident that the estimate SCP algorithm

has shorter execution time compared to the estimate SHLEFT algorithm for a majority of the DAGs. However, the execution times of the estimate SCP and estimate SHLEFT are similar to each other for the mean value analysis, random, and simple fork-join DAGs. The estimate HLEFT and SCP have shorter execution times than the SETF algorithm. Note, however, that the execution time of the estimate SETF is less than twice the execution time of the fastest estimate LS algorithm compared.

Table 5.7 Relative Execution Times of the Estimate LS Algorithms

		Relative Execution Times of Estimate Method Algorithms		
DAG Structure	**Size:**	**SHLEFT**	**SETF**	**SCP**
FFT	Large	1.319	1.230	1.000
HFJ	Small	1.184	1.344	1.000
	Medium	1.251	1.220	1.000
	Large	1.348	1.285	1.000
MVA	Small	1.003	1.026	1.000
	Medium	1.000	1.429	1.069
	Large	1.000	1.364	1.181
OUT	Small	1.191	1.553	1.000
	Medium	1.264	1.980	1.000
	Large	1.395	1.903	1.000
RND	Small	1.000	1.307	1.028
	Medium	1.000	1.575	1.063
	Large	1.000	1.425	1.045
SFJ	Small	1.000	1.371	1.044
	Medium	1.066	1.237	1.000
	Large	1.005	1.198	1.000
All	Small	1.032	1.281	1.000
	Medium	1.081	1.334	1.000
	Large	1.093	1.276	1.000

5.1.4 Trading QoS for Performance using LS algorithms

In this section, the extent to which reductions in required probability of meeting end-to-end deadlines translates into reduction in schedule lengths is analyzed for the best LS algorithm for each DAG. Figures 5.35 - 5.38 plot the average schedule compression metric grouped by DAG structure type, task weight distribution type, DAG CCR, and DAG size, respectively. These figures show that, on average, if missed deadlines can be tolerated even for a relatively small percentage of time, $1E^{-10}$ in particular, the length of the completion time interval can be reduced by over 50%. Tolerating missed deadlines $1E^{-4}$ percent of the time results in a compression of over 60% in general.

Reducing the required probability of meeting deadlines further has diminishing benefit in terms of resulting compression. This is because the probabilities that the schedule will require the time slots near the upper bound of the end-to-end completion time PDF are very small and gradually increase with distance (*i.e.*, time slots) below the upper bound. An examination of the schedules reveals that the probabilities near the upper bound of the completion PDF to be $1E^{-300}$ or lower. In some cases, the probabilities are sufficiently small so as to be reduced to 0.0 by the compiler and/or processor. Therefore, as long as small probability of missing deadlines can be tolerated, the time slots with infinitesimally small probabilities of being required can be removed from the schedule.

The figures also show that while the structure of the DAG does not have a significant impact on the achieved compression, the probability distribution of the task weights affects the amount of compression. Because the exponential and beta

distribution have shapes such that the probabilities near the upper limit of the weight PDF are small, the completion time PDF of schedules for DAGs with these probability distributions also have small probabilities near its upper limit. This results in relatively large compression from small reductions in required probability of meeting end-to-end deadlines.

Conversely, because the shape of the random distribution type is more uniformly weighted, the compression amounts resulting from small reductions in the probability of meeting end-to-end deadlines for DAGs with random task weight distributions are more modest. In general, the maximum compression achievable for the DAGs with exponential, beta, and random distributions is approximately 91%, 75%, and 40%, respectively, given a required probability of meeting end-to-end deadlines of 95%.

Figures 5.37 and 5.38 show that the CCR and size of the DAG have relatively little impact on the shape of the compression metric curve for the DAGs tested for the lower probabilities of meeting end-to-end deadlines (*i.e.*, the compression metric is nearly identical for all probability values across all the DAG sizes).

Note that for these experiments the probability for meeting the deadline is bounded from below by 70% because real-time systems typically require a higher probability of meeting deadlines than 95% to be useful [18].

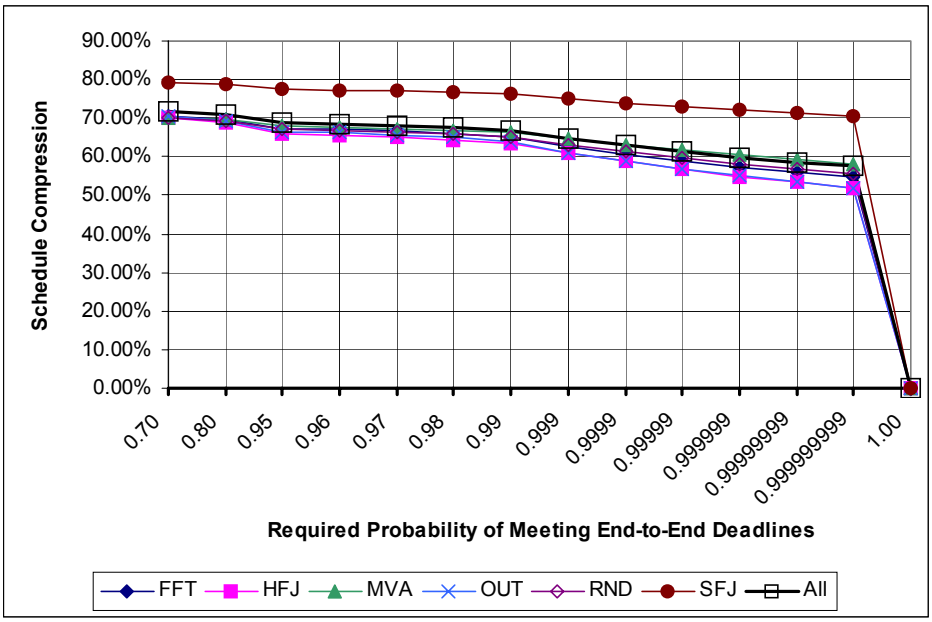


Figure 5.35 LS Compression Grouped by DAG Structure

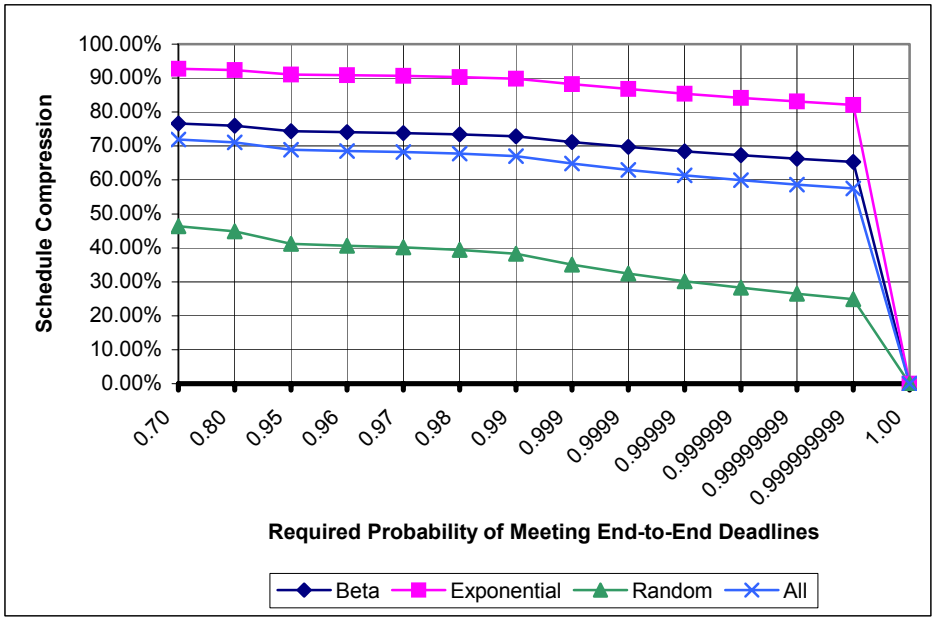


Figure 5.36 LS Compression Grouped by Weight Distribution

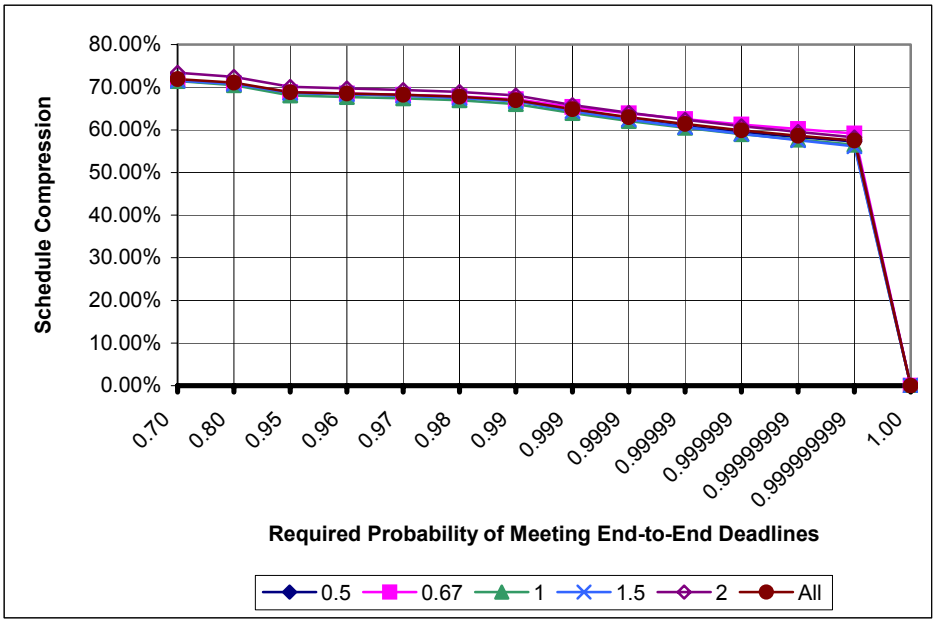


Figure 5.37 LS Compression Grouped by DAG CCR

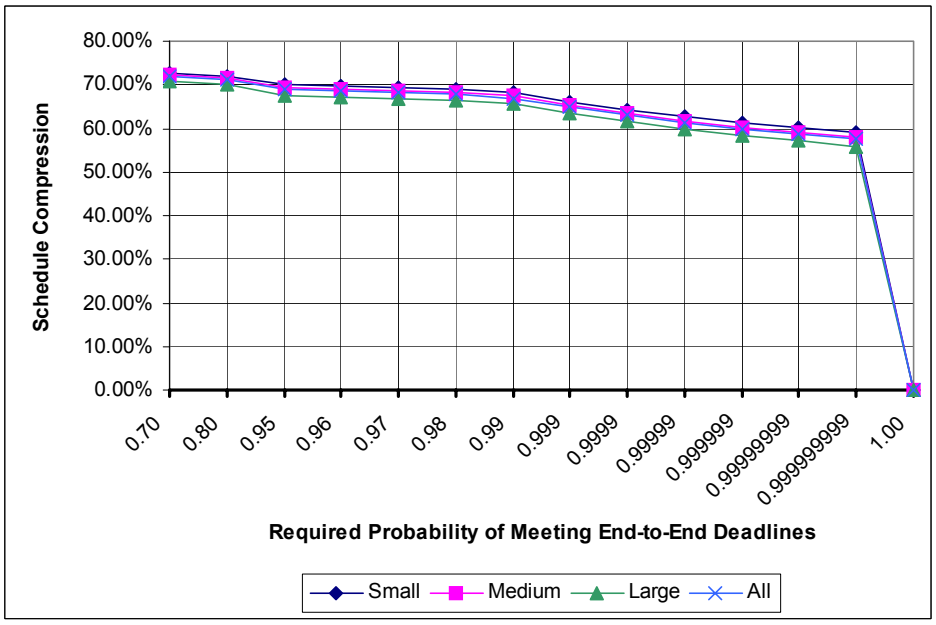


Figure 5.38 LS Compression Grouped by DAG Size

Figures 5.39 - 5.42 plot the QoS-performance tradeoff metric for the DAGs grouped by DAG structure type, task weight distribution type, DAG CCR, and DAG size, respectively. These charts reinforce the notion that reducing the required probability of meeting end-to-end deadlines relative the WCET requirements can result in significant dividends in terms of reduced schedule lengths. Furthermore, as expected, the tradeoff metric rapidly declines to a value approaching 1.0 when the required probability of meeting the deadline is reduced below 99%. Recall that the probabilities (of being used) of the individual time slots at or near the upper limit of the schedule's completion time PDF are much smaller that the probabilities of time slots near the middle of the PDF's defining interval. Note that there is virtually no variance in the QoS-performance tradeoff value across the various DAG characteristics for a specific end-to-end completion probability.

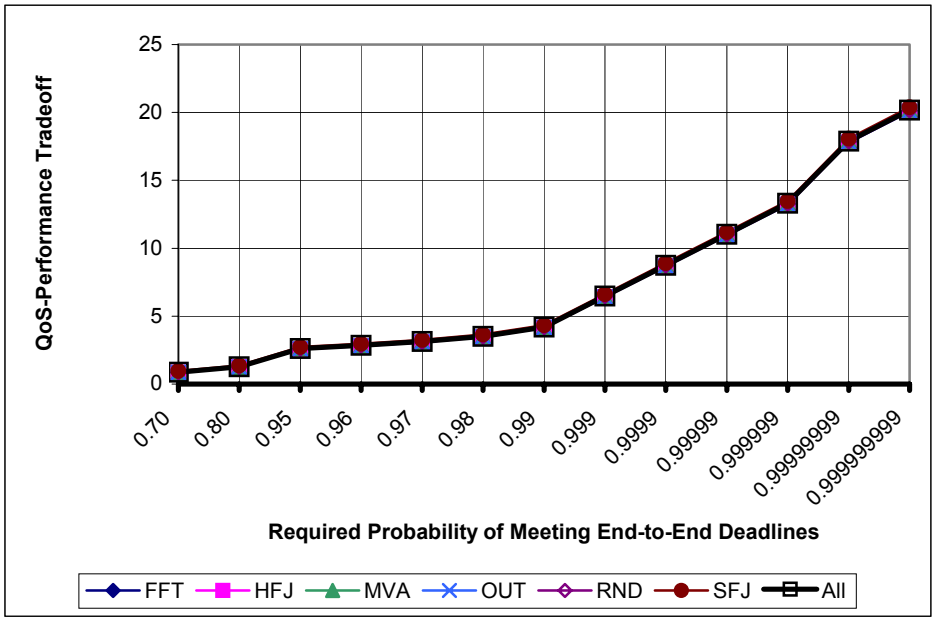


Figure 5.39 LS QoS-performance Tradeoff Grouped by DAG Structure

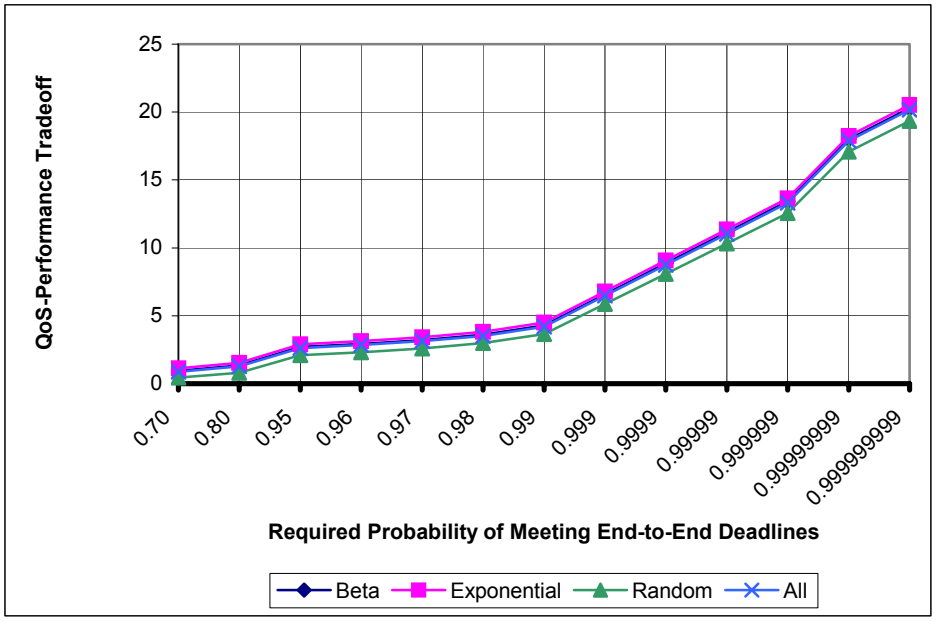


Figure 5.40 LS QoS-Performance Tradeoff Grouped by Weight Distribution

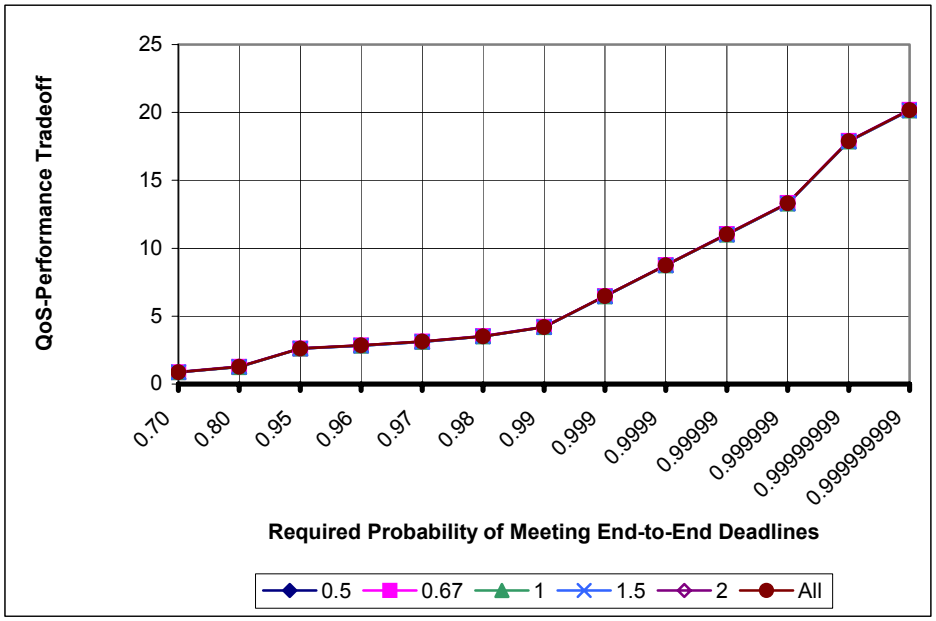


Figure 5.41 LS QoS-Performance Tradeoff Grouped by DAG CCR

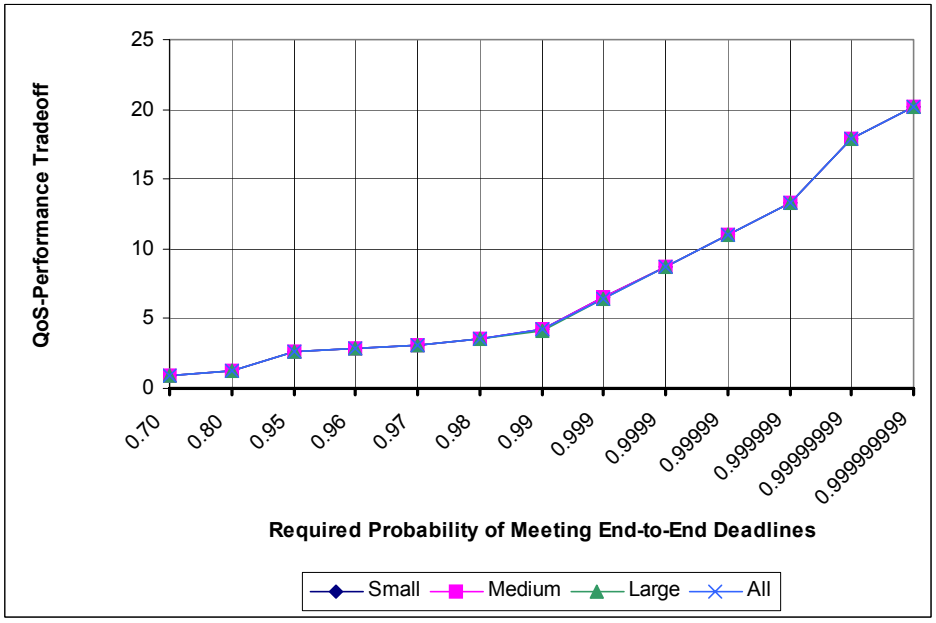


Figure 5.42 LS QoS-Performance Tradeoff Grouped by DAG Size

5.1.5 Stochastic Jitter Control with LS

The following series of figures plots the reduction in the schedules' completion time stochastic jitter for specific values of the jitter control parameter specified to the best LS algorithm for each DAG. Recall that the jitter control parameter prevents the scheduler from starting tasks as early as possible, and instead, delays the tasks so as to prevent the task completion PDF of the previously executing task from lengthening the defining interval of the completion time PDF of the new task.

Figures 5.43 - 5.46 plot the stochastic jitter factor averaged over all DAGs grouped by DAG structure type, task weight distribution type, DAG CCR, and DAG size, respectively. These figures show the relatively high stochastic jitter inherent in the schedules produced for the various DAGs. Specifically, the HFJ and SFJ DAGs exhibit significant amount of completion time jitter whereas the FFT, OUT, and Random DAGs exhibit below average jitter. The MVA DAGs exhibit average jitter.

Other DAG characteristics (*i.e.*, task weight distributions, CCRs and sizes), on average, have relatively little impact on the jitter inherent in the schedules, compared with the impact of DAG structure. Therefore, the average stochastic jitter factor curves observed across these characteristics are close to each other. Note that using a jitter control parameter of 25% results in a jitter factor of less than 5. Increasing the jitter control parameter beyond 25% has limited impact on the schedule's stochastic jitter.

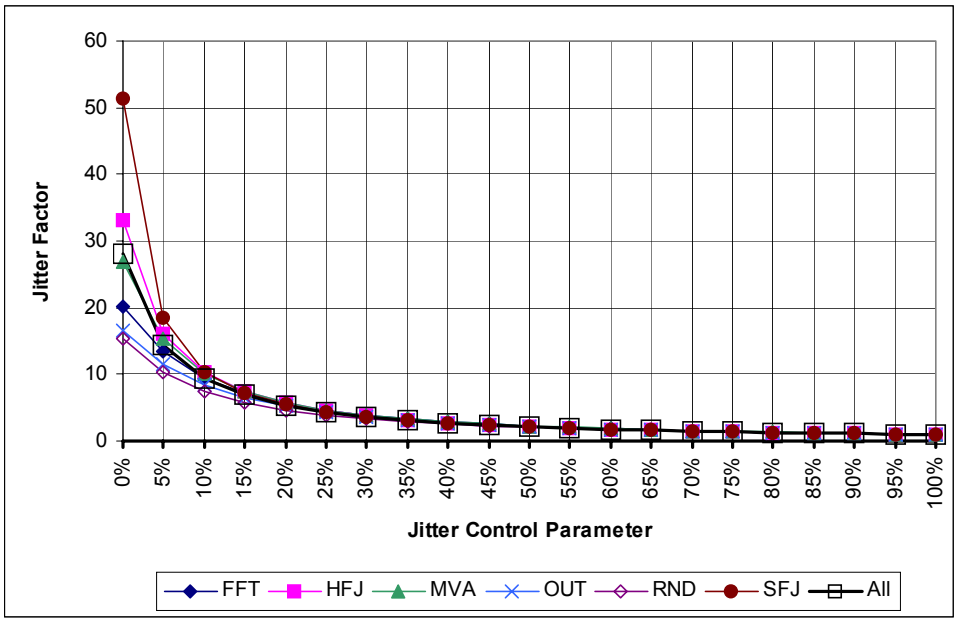


Figure 5.43 LS Jitter Control Factor Grouped by DAG Structure

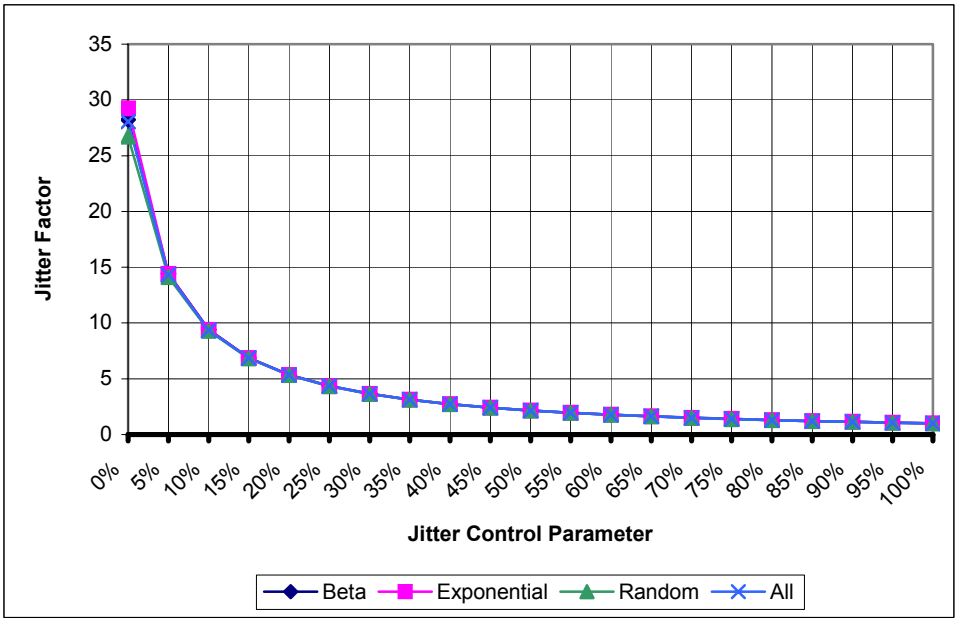


Figure 5.44 LS Jitter Control Factor Grouped by Weight Distribution Types

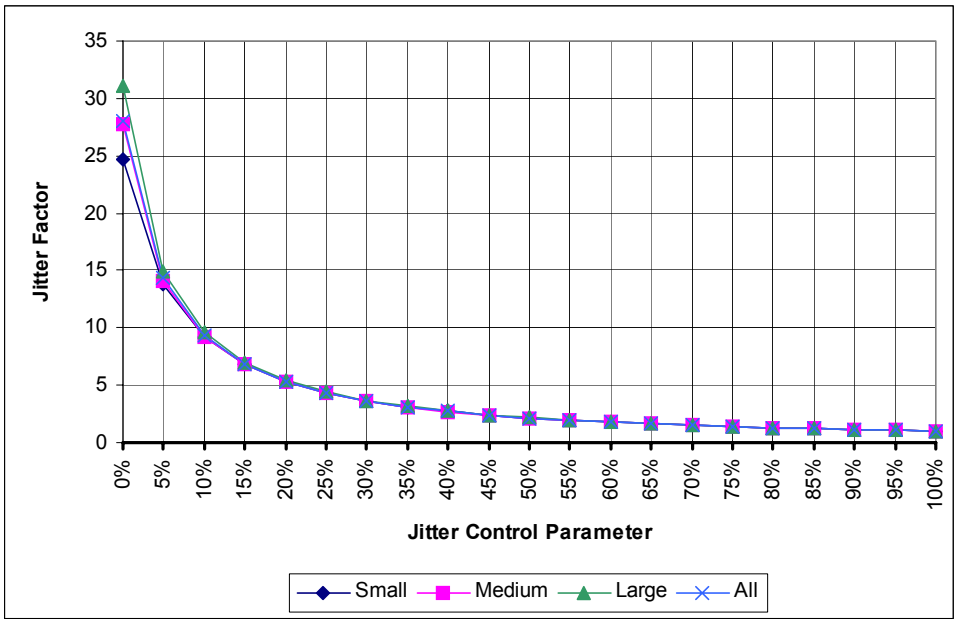


Figure 5.45 LS Jitter Control Factor Grouped by DAG Size

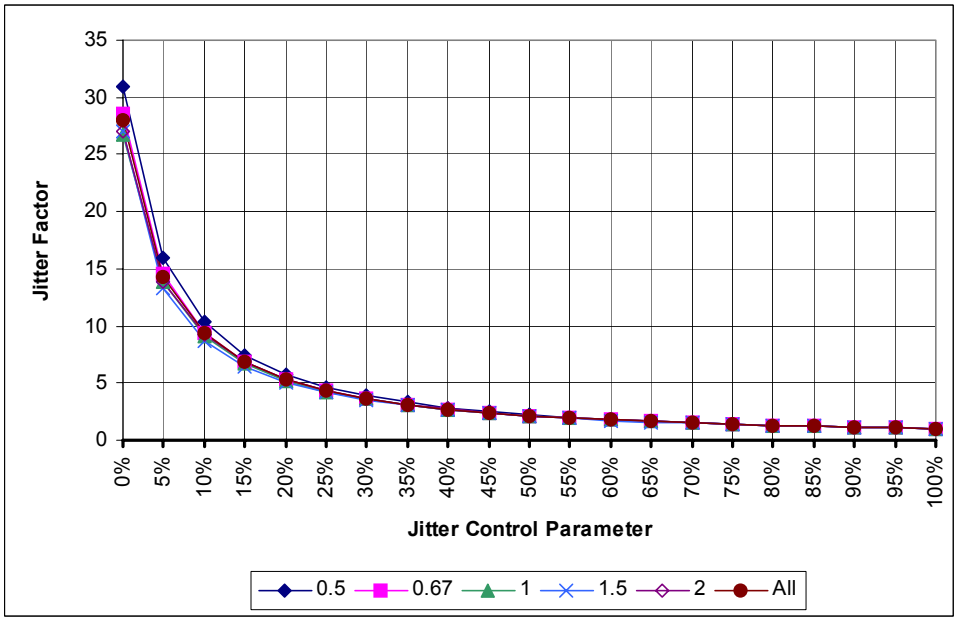


Figure 5.46 LS Jitter Control Factor Grouped by DAG CCR

Figures 5.47 - 5.50 plot the stochastic utilization that results from changing the jitter control parameter averaged over all DAGs grouped by the DAGs' structure types, weight distribution types, sizes, and CCRs, respectively. These figures show that the stochastic utilization increases with an increase in the jitter control parameter. The increase in utilization occurs because an increase in the jitter control parameter decreases the footprint of the resulting schedule.

These figures indicate that the schedules for the HFJ and SFJ structure types have inherently low stochastic utilization (*i.e.*, less than 50% when no jitter control is applied) whereas the other structure types have higher stochastic utilization. Note the close correspondence with the Jitter Factor chart in 5.43 where the schedules for the HFJ and SFJ structures display higher jitter than the scheduled of the other DAG structure types. The CCR and weight distribution characteristics of DAGs have a small impact on the stochastic utilization metric of the schedules. Conversely, a DAG's size has a relatively small impact on the stochastic utilization metric of the resulting schedules.

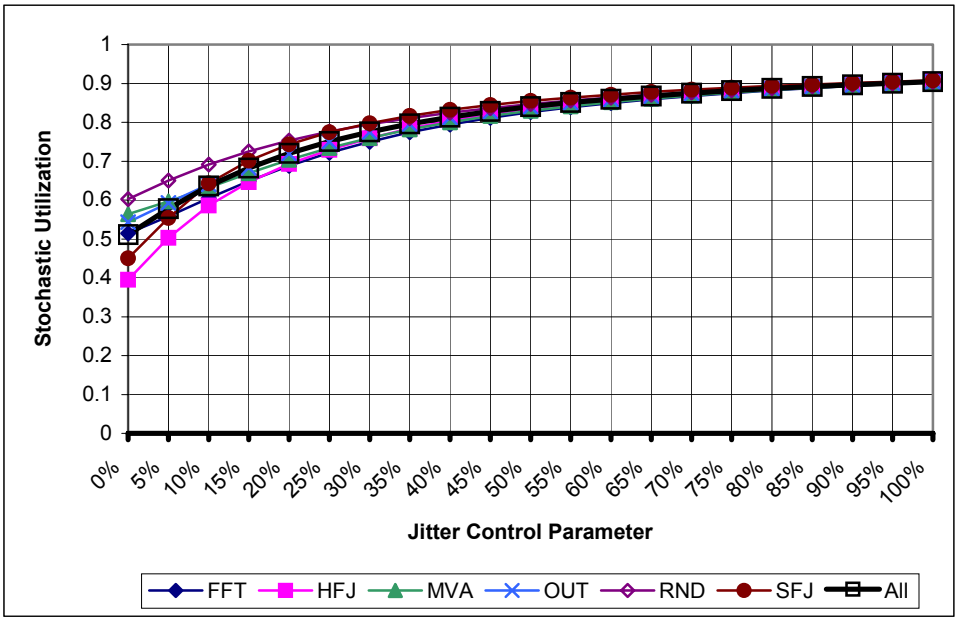


Figure 5.47 LS Utilization Grouped by DAG Structure

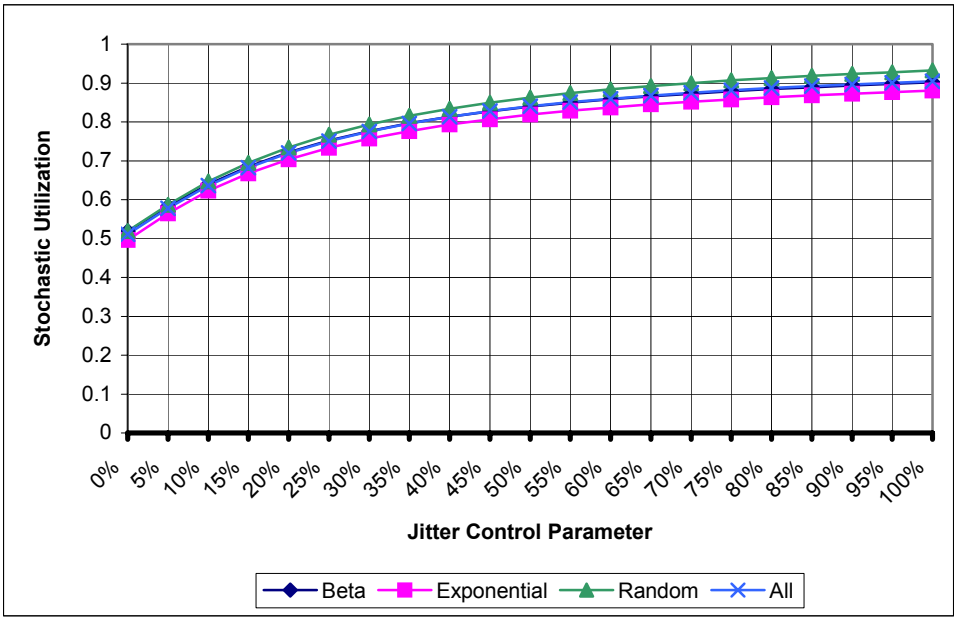


Figure 5.48 LS Utilization Grouped by Weight Distribution

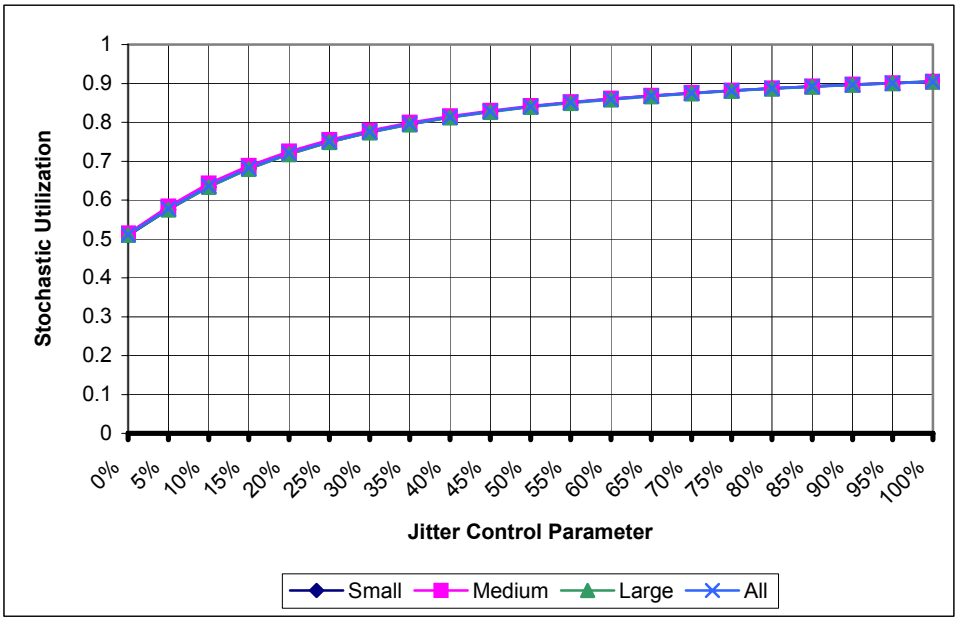


Figure 5.49 LS Utilization Grouped by DAG Size

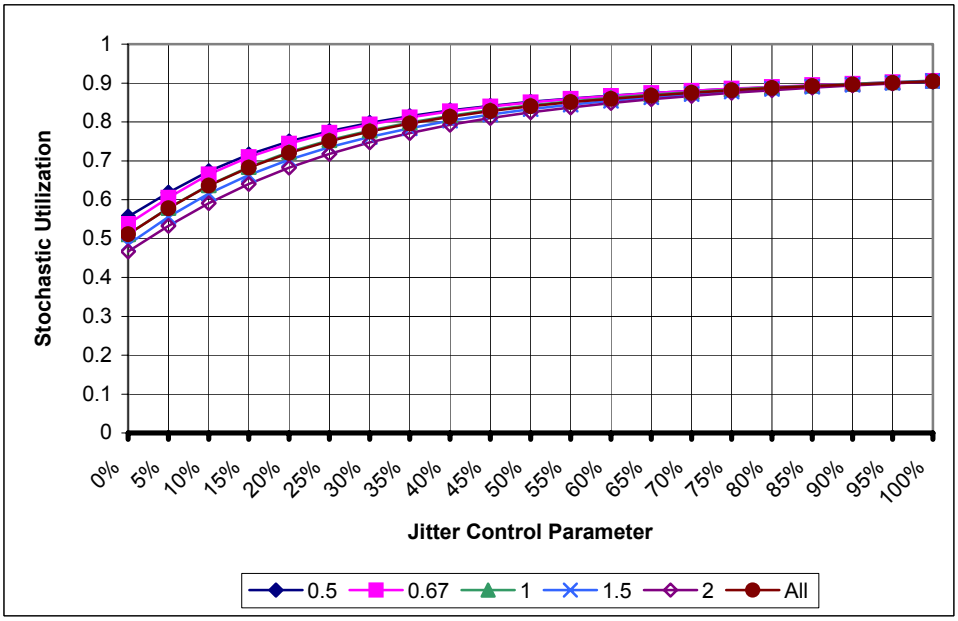


Figure 5.50 LS Utilization Grouped by DAG CCR

5.1.6 Schedule Compression versus Jitter Control with LS

Reducing stochastic jitter by delaying the start time of tasks is a simple approach for controlling the width of the schedule's completion time PDF interval. Intuitively, however, it would appear that delaying tasks should negatively impact the ability of the scheduling algorithms to reduce the required probability of meeting the end-to-end deadlines in order to reduce schedule lengths.

Figure 5.51 plots the schedule compression metric in response to changes in the required probabilities for meeting end-to-end deadlines for specific jitter control parameter values averaged over all DAGs. The figure shows that the maximum compression increases when moderate amount of jitter control is applied. However, compression decreases when the jitter control parameter is increased to values above 0.50.

The increase in schedule compression when moderate jitter control is applied is explained as follows. Delaying tasks by a small amount of time has minimal impact on the completion time PDF of the task (especially near the upper bound of the completion PDF interval) because the individual probabilities that the task will begin at the time units in the vicinity of the lower bound of the interval defining the start time PDF of the task are relatively small.

Therefore, even after applying a small amount of delay, reducing the required probability of meeting end-to-end deadlines results in the reduction of the schedule length by approximately the same number of time units compared to when there is no delay. In other words, the numerator of equation (4.8) (*i.e.*, $u_{fschedule} - M(x) + 1$) remains

relatively constant whether or not a small amount of jitter is applied in combination with a specific probability of meeting end-to-end deadline.

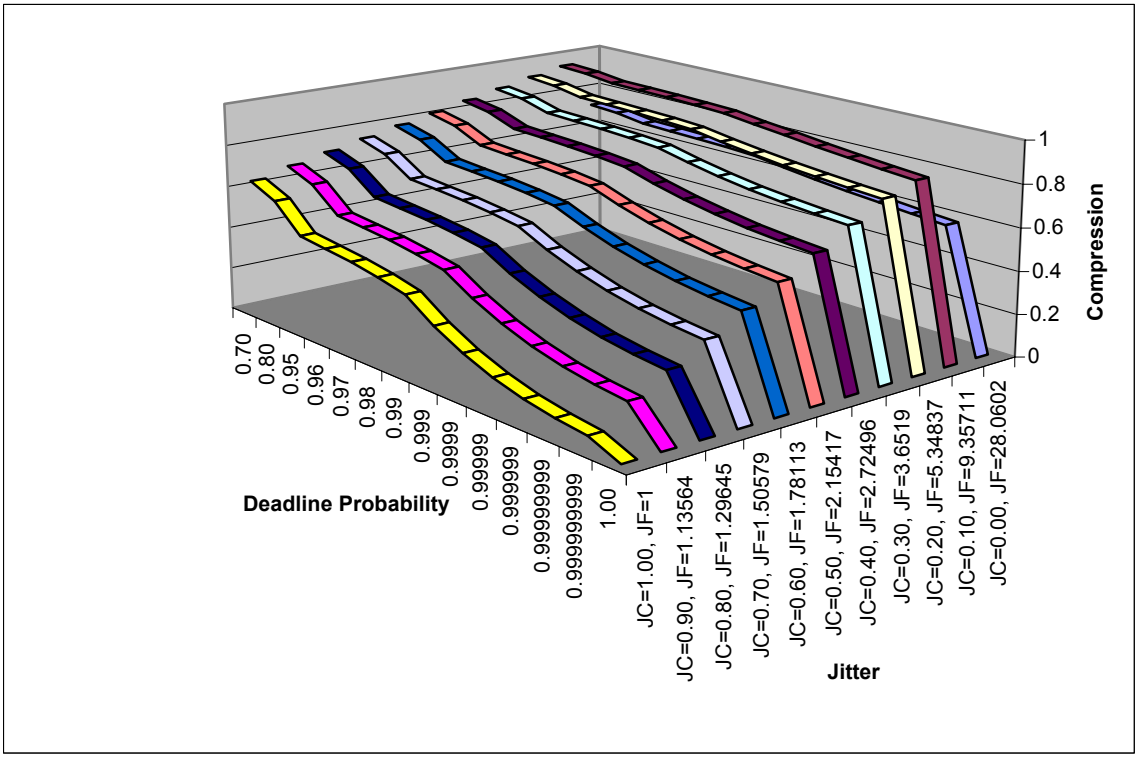


Figure 5.51 LS Compression vs. Jitter Control Factor for All DAGs

However, because the lower bound of the completion time interval is increased because of the delay introduced by jitter control, the total width of the completion time PDF interval is shortened. In other words, the denominator of equation (4.8) (*i.e.*, $u_{fschedule} - l_{fschedule} + 1$) is reduced. Therefore, when a small amount of jitter control is applied, the compression metric reports greater compression as compared to the case when no jitter control is applied.

Applying relatively large amounts of jitter control (*i.e.*, significantly delaying tasks), on the other hand, deforms the completion time PDF such that the individual probabilities of the time slots near the upper bound of the completion time PDF being needed are increased. This deformation can be viewed a partial translation of the PDF towards the higher domain values of the PDF. This deformation implies that decreasing the required probability of meeting end-to-end deadlines will not result in as significant a reduction in the completion time of the schedules as compared to when no jitter control or small amount of jitter control is applied. Therefore, specifying a large value of the jitter control parameter results in lower schedule compression values.

5.2 Stochastic Genetic List Scheduling Approach

This section presents and analyzes the performance of the GLS algorithm and compares the schedules created by the GLS approach with the schedules created by the best LS schedules. The GLS algorithm takes a large amount of time to construct the schedule. Therefore, in the interest of completing this research within a reasonable amount of time, schedules for only those DAGs with the FFT structure having a CCR of 1.0 are constructed and analyzed below. This implies that schedules for a total of 15 DAGs were created using the GLS approach. Also recall that in order to decrease the schedule construction time, the GLS uses the two phased scheduling approach similar to that of the estimate LS algorithms.

The GLS approach investigated here uses the tasks' WCET for the estimate value. This choice of estimate value is justified by the results summarized in Figure 5.4. Recall that this figure plots the improvement in maximum schedule length averaged across all

estimate algorithms, grouped by DAG structure type, and shows that the FFT DAG type has an improvement of at most 0.75% when estimate values other than WCET are used.

5.2.1 Comparison of Stochastic LS and Stochastic GLS

Table 5.8 lists and compares the performance of the GLS and LS approaches for the various FFT DAGs. The values in the "Length of Best LS Schedule" columns are the maximum schedule lengths for the best schedule for each of the DAGs using any combination of LS heuristic and algorithm control parameter (*i.e.*, weight estimate or slot-fitting threshold value). The GLS approach produced shorter schedules than the LS approach for 11 of the 15 DAGs. Furthermore, for 7 of these 11 DAGs, GLS produced significantly shorter schedules than the best corresponding LS schedule (*i.e.*, shorter by at least 9.69%). The average improvement in the maximum schedule lengths resulting from the use of the GLS approach as compared to the LS approach over all the DAGs is approximately 12.4%. By contrast, in the four cases that LS outperforms GLS, the GLS schedule is at most 1.62% longer than the LS schedule.

Table 5.8 Comparison of GLS and LS Schedules for FFT DAGs

Distribution Type	CCR	Length of GLS Schedule	Length of Best LS Schedule	Improvement in Schedule Length
Beta	0.5	23,231	41,556	44.10%
	0.67	18,587	22,612	17.80%
	1.0	13,111	14,518	9.69%
	1.5	15,101	15,037	-0.43%
	2.0	18,875	19,257	1.98%
Exponential	0.5	23,024	35,597	35.32%
	0.67	18,554	23,908	22.39%
	1.0	13,373	13,528	1.15%
	1.5	16,471	17,144	3.93%
	2.0	18,861	19,709	4.30%
Random	0.5	22,857	34,260	33.28%
	0.67	17,472	20,686	15.54%
	1.0	13,581	13,432	-1.11%
	1.5	15,349	15,298	-0.33%
	2.0	17,949	17,663	-1.62%
			Average Improvement:	12.4%

The GLS algorithm, however, takes several hours to construct a schedule on a cluster of modern processors (eight 2.4MHz Xeon processors with 1GB RAM interconnected using gigabit Ethernet) as opposed to the few seconds it takes the exact SHLEFT and SCP algorithms, or the few minutes it takes the exact SETF algorithm. If time is a premium, it may not be possible to use the GLS to produce shot schedules. However, when time permits, the GLS approach should be utilized along with the LS approach. The LS approach will consume only a fraction of time as compared to GLS, but may provide better schedules than GLS for some small portion of the problem space.

5.2.2 Trading-off Performance for QoS with GLS

Figure 5.52 plots the schedule compression achieved by using the GLS approach for the FFT DAGs grouped by the task weight distribution types. This chart shows that significant schedule compression can be achieved when the GLS approach is used to construct schedules. Similar to the compression observation with LS approach (and for the same reasons) in Figure 5.36, DAGs with the exponential and beta task weight distributions exhibit large compression values of over 90% and over 79%, respectively.

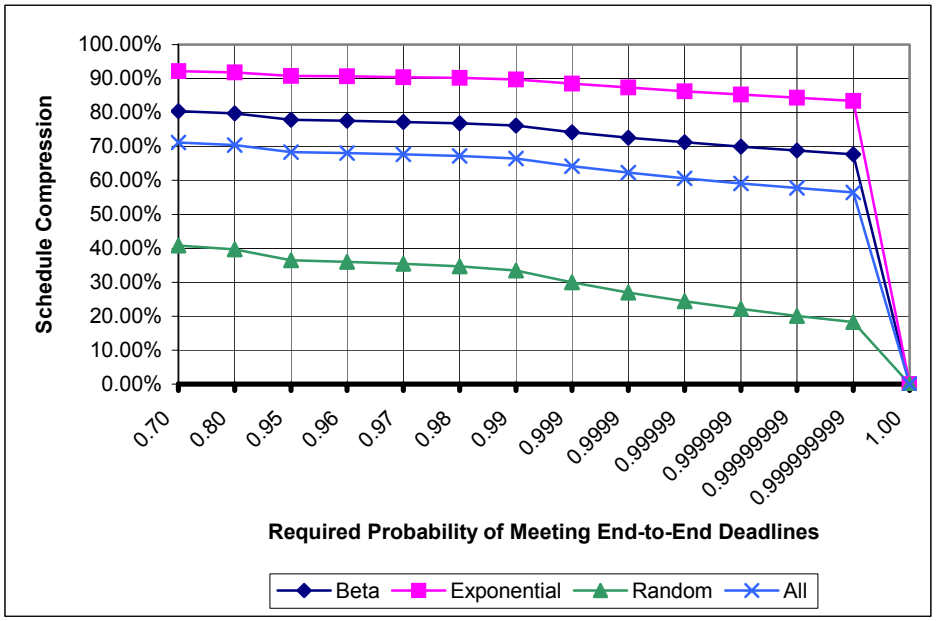


Figure 5.52 GLS Schedule Compression Grouped by Weight Distribution

Figure 5.53 plots the schedule compression achieved by using the GLS approach for the FFT DAGs grouped by DAG CCR. This chart is similar to the LS compression result grouped by CCR, in Figure 5.37. The DAG CCR has relatively little impact on schedule compression.

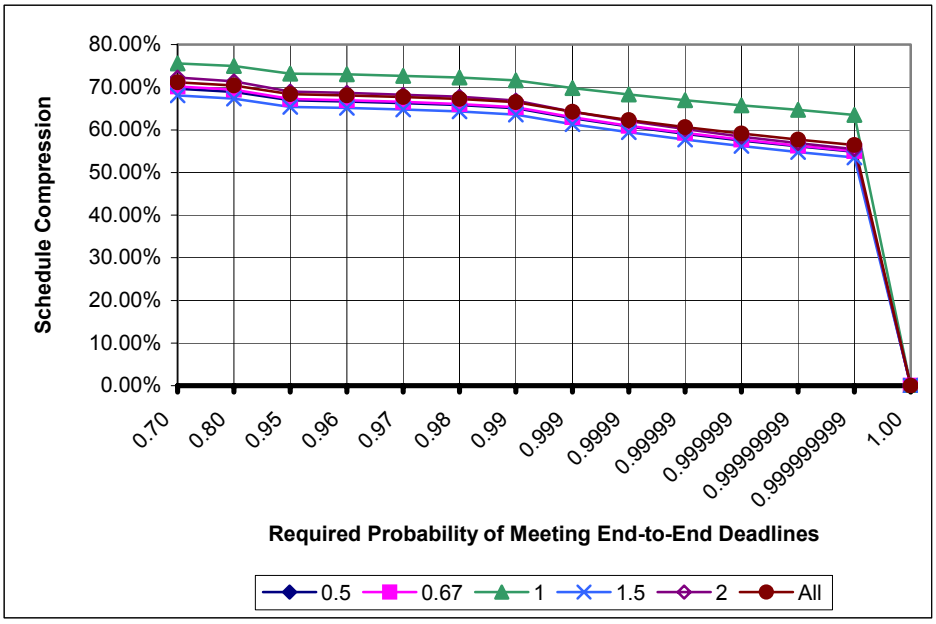


Figure 5.53 GLS Schedule Compression Grouped by DAG CCR

Figures 5.54 and 5.55 plot the QoS-performance tradeoff metric for the schedules produced by the GLS approach for the FFT DAGs, grouped by task weight distribution type and DAG CCR. These charts support the idea that reducing the required probability of meeting end-to-end deadlines relative to the WCET requirements can result in significant dividends in terms of reduced schedule lengths. Furthermore, as expected, the tradeoff metric rapidly declines to a value approaching 1.0 when the required probability of meeting the deadline is reduced below 99%.

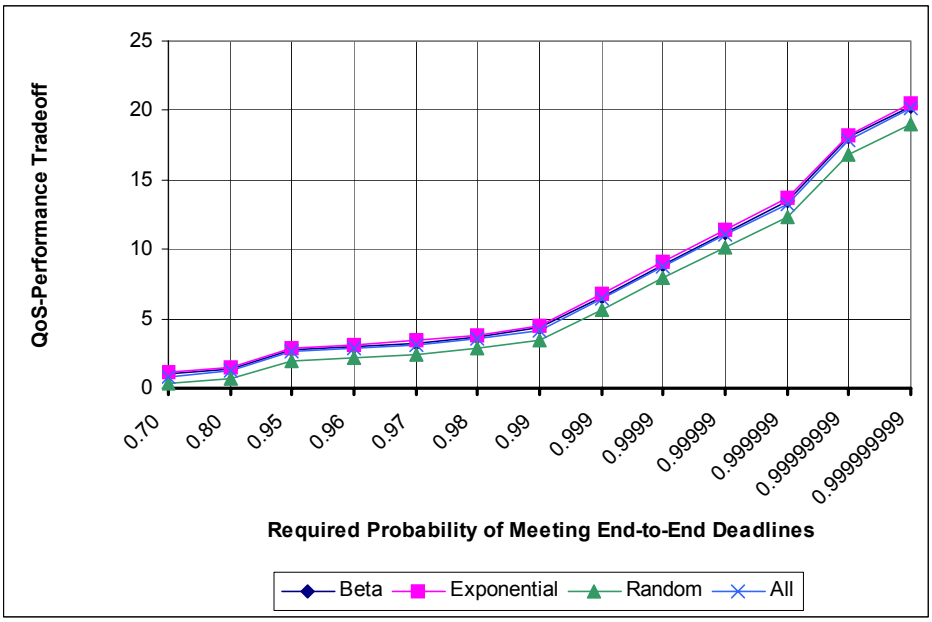


Figure 5.54 GLS QoS-Performance Tradeoff Grouped by Weight Distribution

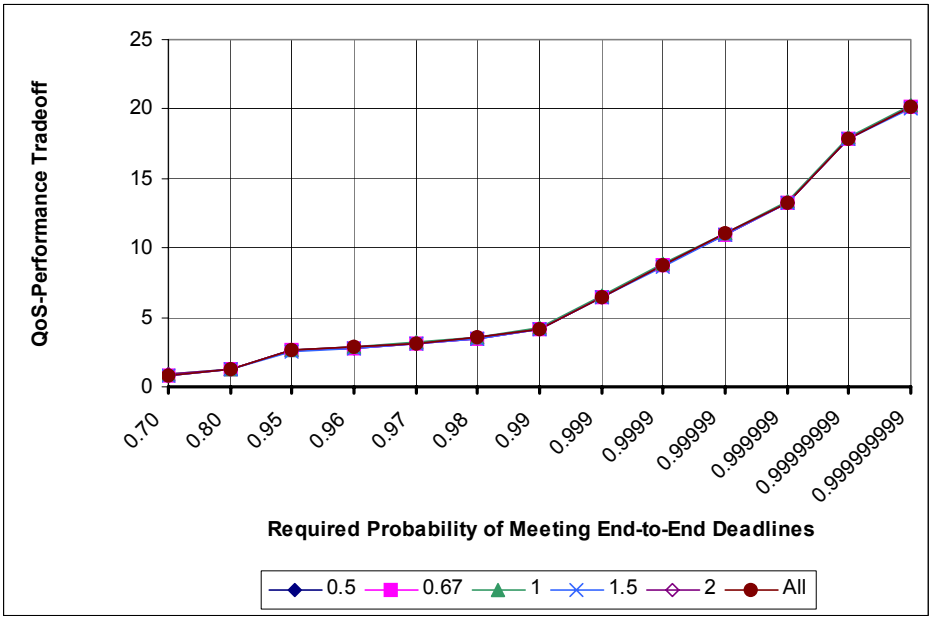


Figure 5.55 GLS QoS-Performance Tradeoff Grouped by DAG CCR

5.2.3 Jitter Control with GLS

Figures 5.56 and 5.57 plot the average stochastic jitter factor grouped by task weight distribution type and DAG CCR, respectively, resulting from specifying various amounts of jitter control to the GLS algorithm for the FFT DAGs. The resulting jitter factor values across corresponding jitter control parameter points are nearly identical to each other for the three task weight distribution types. The jitter factor curves for the various DAG CCRs are also relatively close to each other. This indicates that different weight distribution types and different DAG CCRs have similar effects on the jitter characteristics of the schedules produced by the GLS algorithm.

A comparison of the jitter factor curves in Figure 5.56 with FFT jitter factor curves in Figure 5.43 indicates that the schedules produced by the GLS algorithm have nearly identical inherent jitter characteristics (*i.e.*, a jitter factor value of 20 when the jitter control parameter value is 0) compared with the best schedules produced for the corresponding FFT DAGs by the LS approach.

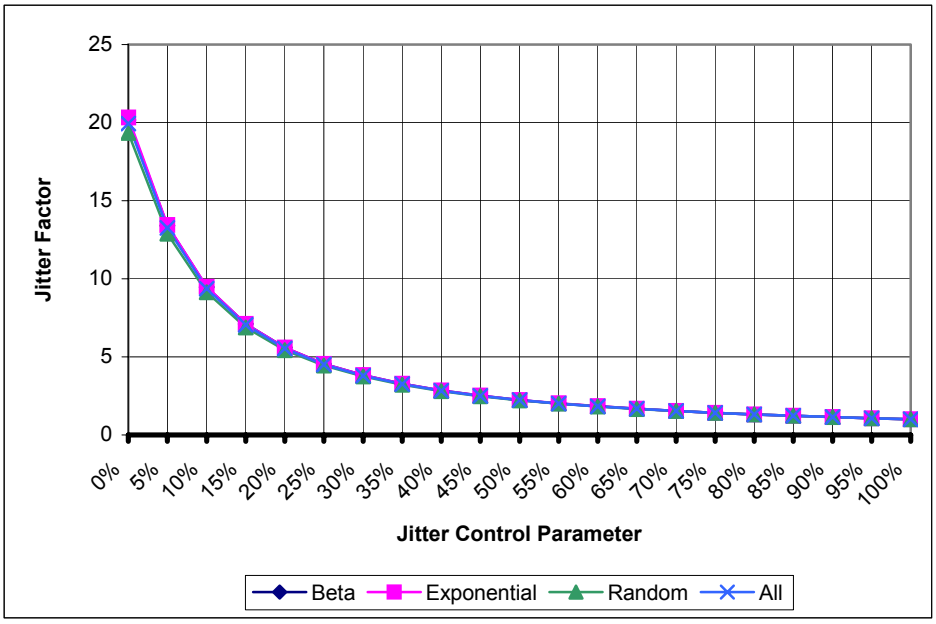


Figure 5.56 GLS Jitter Control Grouped by Weight Distribution

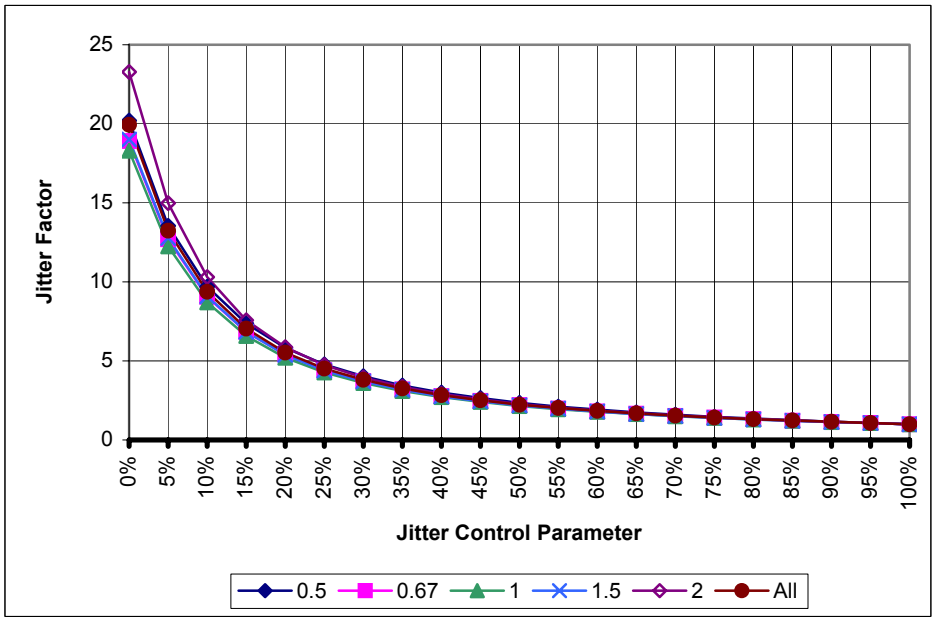


Figure 5.57 GLS Jitter Control Grouped by DAG CCR

Figures 5.58 and 5.59 plot the stochastic utilization of resources in the schedules produced for the FFT DAGs by the GLS algorithm. These charts show that the weight distribution and DAG CCR have little impact on the stochastic utilization in the schedules produced by the GLS algorithm.

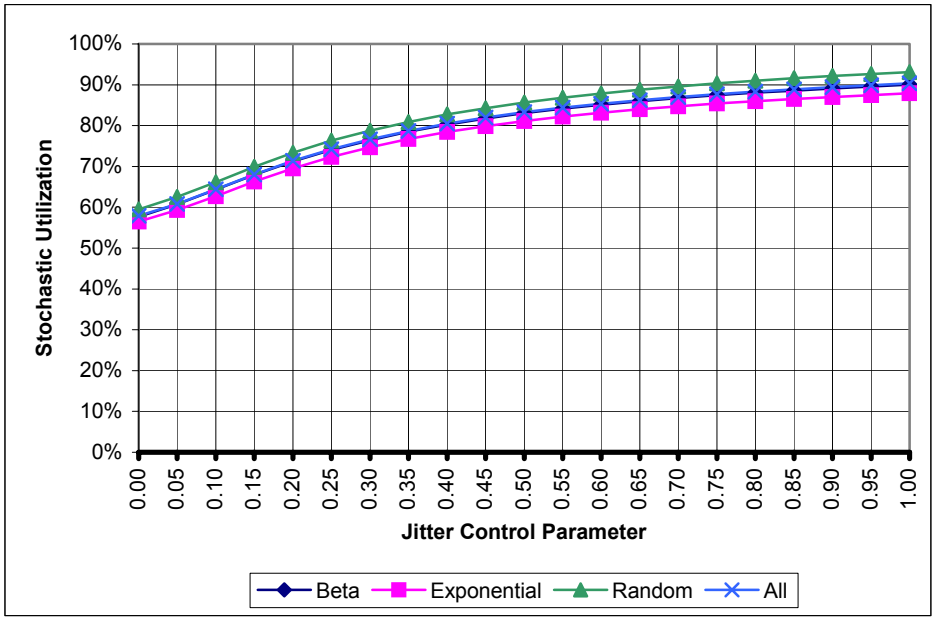


Figure 5.58 GLS Utilization Grouped by Weight Distribution

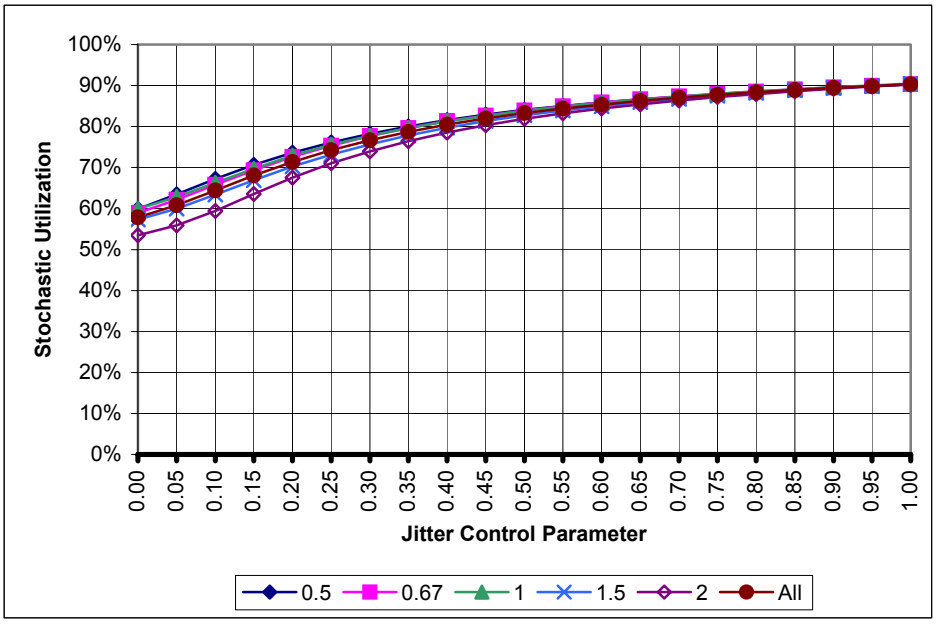


Figure 5.59 GLS Utilization Grouped by DAG CCR

5.2.4 Schedule Compression versus Jitter Control with GLS

Figure 5.60 plots the schedule compression metric in response to changes in the required probabilities for meeting end-to-end deadlines for specific jitter control parameter values averaged over all DAGs. The figure shows that the maximum compression increases when moderate amount of jitter control is applied. However, compression decreases when the jitter control parameter is increased to values above 0.50. This result is similar to the result in Figure 5.51 showing the compression in response to the jitter control parameter for LS algorithms.

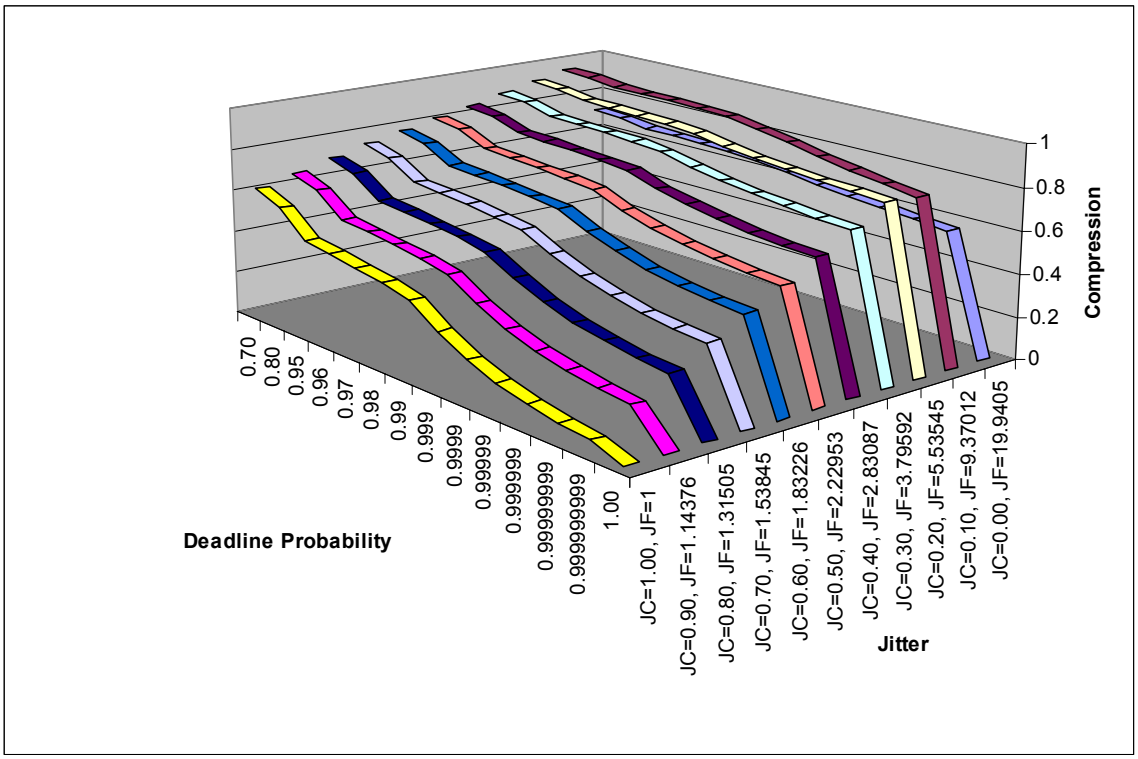


Figure 5.60 GLS Compression vs. Jitter Control Factor

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the contributions and results of this dissertation and presents a variety of potential extensions to this research.

6.1 Contributions and Results

This dissertation research makes several contributions to the state of the art in real-time scheduling. The primary contribution is the generalization of the traditional deterministic LS and GLS approaches in order to create non-preemptive soft real-time schedules for parallel applications consisting of tasks with varying task execution time requirements and with inter-task precedence constraints. The parallel real-time applications are modeled as DAGs with vertices and edges representing computation tasks and communication/synchronization tasks, respectively. Task weight PDFs model the variable execution time requirements of the tasks in the application. In this research, the variations in the task weights are assumed to be independent between tasks.

Deterministic LS and GLS algorithms typically manage contention for processor resources while ignoring communication contention. The stochastic LS and GLS algorithms and heuristics developed in this research are novel in that they also account for contention that occurs when communication tasks compete with each other for access to finite-capacity processor-to-network links. Essentially, scheduling decisions for

vertices and edges play an equally important role in determining the QoS and performance characteristics of the stochastic schedules produced by the techniques developed here.

Innovative algebra for using task weight PDFs, as opposed to fixed weights, for constructing schedules is developed as part of this dissertation. These operators are used to compute the maximum and minimum PDFs from sets of independent PDFs in order to compute the starting time and ending time PDFs of idle slots in the schedule into which tasks are allocated. Convolution is used to compute a task's completion time PDF from the task's start time PDF and weight PDF.

In order to determine whether a task can be inserted before another, previously scheduled task, without causing substantial perturbation to the existing scheduling decisions, the innovative slot-fitting threshold heuristic is introduced. This heuristic requires the computation of the probability that the task to be inserted can complete before the planned start time of the subsequent task. An algorithm to reflect the effect of delay on the start time PDF of tasks is also developed in order to mitigate the stochastic jitter that occurs in the soft real-time schedules.

The appropriate use of these PDF operations in LS and GLS algorithms is also another significant contribution of this dissertation. Because the convolution, minimum, and maximum operators, as developed here, are valid only over independent PDFs, this dissertation clearly describes scheduling situations when the PDF manipulations are applicable, and situations where alternative techniques are to be applied.

In order to reduce the time taken to construct schedules, the dissertation develops the idea of using fixed estimated task weights while making initial scheduling decisions and then computing the final schedule completion PDF from the initial schedule, as opposed to using exact PDFs at every scheduling step. While the estimate approach is faster than the exact approach, the estimate approach produces poorer schedules than the exact approach for nearly 62% of the sample DAGs. However, because the algorithms based on the estimate method are significantly faster than the algorithms based on the exact method, both methods can be used to construct schedules for a DAG and the shortest schedule can be used as the solution.

Experimental results show that using WCET and best-case execution time for task weight estimates produce schedules with roughly equal lengths. Using the weights at which the tasks' probabilities of completion range from 50% to 70% as weight estimates (*i.e.*, near the expected value of the edge weights) results in schedules that are over 20% longer, in general, than the schedules that are produced when the tasks' WCET are used as weight estimates.

The primary reason why the LS algorithms using the exact method are able to produce significantly shorter schedules than the estimate method algorithms is the ability of the exact algorithms to exploit the slot-fitting heuristic. Experimental results show that a slot-fitting threshold of 95% produces an average improvement in overall schedule length of over 6% compared to using a slot-fitting threshold of 100% (*i.e.*, disallowing the exploitation of the slot-fitting heuristic). Experiments also show that reducing the slot-fitting threshold further has diminishing returns in terms of reducing schedule

lengths. Using a slot-fitting threshold of 60% produces an average of slightly over 7% reduction in schedule lengths.

An implementation of a steady state, parallel, genetic list scheduling algorithm using the island communication model is also developed and investigated as part of this dissertation. The GLS algorithm is used to construct even shorter schedules that those produced by the LS algorithm. As in the LS algorithms, the GLS algorithm also uses the PDF manipulation operators to evaluate potential schedules. In order to reduce the execution time of the GLS algorithm, the estimate method is used to construct the candidate schedules from the chromosomes. The genetic representation, genetic operators, and chromosome migration patterns are an amalgamation of techniques developed by a variety of researchers and many of the parameters controlling the GLS were derived from precursor research into constructing deterministic schedules for DAGs [41]. These GA techniques and GLS parameters were adapted specifically for producing stochastic schedules for this dissertation.

The dissertation shows empirically that the hypothesis is valid. The computation of the completion time PDF of the schedule allows real-time systems designers to systematically tradeoff QoS (probability of meeting end-to-end deadline and jitter) for schedule length. For the wide variety of DAGs tested, results show that, on average, the length of the schedule completion time interval can be reduced by 30% if the required probability of meeting end-to-end deadlines is reduced from 100% to 99.99999999% (*i.e.*, a reduction in probability of $1E^{-10}$). Similarly, reducing the required probability of

meeting end-to-end deadlines to 99.99% results in a 60% reduction, on average, of the length of the completion time interval.

Experimental results also show how the completion time PDF and schedule resource utilization are affected when tasks' start times are delayed in order to reduce stochastic jitter. In general, stochastic utilization improves to over 88% when a jitter control parameter of 75% is specified. However, tight control of jitter reduces the ability to tradeoff the schedule's probability of meeting end-to-end deadlines for reduced schedule length. In general, using a jitter control parameter greater than 25% has an increasingly significant impact on this ability to trade QoS for schedule length.

6.2 Future Work

A number of extensions are possible to the research conducted as part of this dissertation. The most restrictive assumption made in the PDF manipulation operators and the scheduling algorithms is that the task computation requirement time PDFs are independent. In many applications, execution times of successive tasks are related to each other because task behavior depends on the characteristics of the data being processed (*e.g.*, data size and/or locality). In other words, observation of a particular execution time requirement of one task has a distinct influence on the observed execution time of subsequent tasks. Therefore, construction of stochastic schedules for these applications will require new PDF manipulation algebra to account for inter-dependent task execution times.

Another possible extension to the stochastic scheduling techniques and algorithms developed as part of this dissertation is to consider applications in which different tasks

have different QoS requirements. In other words, instead of only considering the probability of completing the entire schedule within an end-to-end deadline, the completion time deadlines for individual tasks (some or all) in the application can be considered when making scheduling decisions. Similarly, different jitter requirements for different tasks can also be considered.

In the algorithms developed for this dissertation, jitter control manipulations are performed after all scheduling decisions are made. However, it is possible that the modifications made to the scheduled tasks' starting and completion time PDFs by the jitter control mechanisms are of sufficient magnitude as to influence the performance of the slot-fitting heuristic. Therefore, an investigation into the interplay between jitter control and the slot-fitting heuristic is required.

The PDF manipulation operators developed in this dissertation can take a long time to compute for large PDFs. Furthermore, memory requirement for manipulating large PDFs is also large. Therefore, in order to mitigate time and storage requirements during construction of stochastic schedules, techniques to manipulate reduced resolution PDFs can be investigated and developed. These techniques can store a lower resolution representation of the PDF and use interpolation techniques to generate the probability values for specific time values. These techniques must also provide techniques for minimizing and estimating the overall margin of error in the stochastic schedule so produced.

In order to speed the GLS algorithm, the estimate method for schedule construction is used during chromosome evaluations. Analysis of the LS algorithms

shows that the algorithms using the exact method often outperform the algorithms using the estimate methods. This suggests that an investigation into the performance of GLS algorithms based on the exact method for schedule construction is required. This investigation will determine if the reduction in schedule lengths (if any) resulting from the use of exact PDF manipulations in the GLS algorithms justify the increased cost in terms of schedule construction time.

Another possible extension specific to the GLS approach is to modify the selection operator to use utilization, or compression, or QoS-Performance tradeoff metrics instead of using schedule length and processors required as the chromosome ranking criteria. This modification may result in schedules with significantly different characteristics than those produced by the GLS algorithm in this dissertation.

REFERENCES

[1] Luca Abeni and Giorgio Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp 3-13, 1998.

[2] Luca Abeni and Giorgio Buttazzo, "QoS Guarantee Using Probabilistic Deadlines," in *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, pp. 242-249, 1999.

[3] Thomas L. Adam, K. Mani Chandy, J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, vol 17, no. 12, pp. 685-690, 1974.

[4] ANSI Standard X3T9.5, *Fiber Distributed Data Interface (FDDI): Token Ring Medium Access Control (MAC)*, 1987.

[5] Manoj Apte, Srigurunath Chakravarti, and Anthony Skjellum, "Time-based Linux for Real-Time NOWs and MPI/RT," In *Proceedings of the IEEE Real Time Systems Symposium*, Phoenix, Arizona, December 1999.

[6] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring, "Scheduling with Implicit Information in Distributed Systems," *Proceedings of the SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pp. 333-243, 1998.

[7] Andrea C. Arpaci-Dusseau, "Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems," *ACM Transactions on Computer Systems*, vol 19, no. 3, 2001.

[8] S. Ashour, A Decomposition Approach for the Machine Scheduling Problem, *International Journal of Production Research*, vol 6, no. 2, pp 109-122, 1967.

[9] Alia Atlas and Azer Bestavros, "Multiplexing VBR Traffic Flows with Guaranteed Application-Level QoS Using Statistical Rate Monotonic Scheduling," in *Proceedings of the 8th IEEE International Conference on Computer Communications and Networks*, 1999.

[10] Alia Atlas and Azer Bestavros, "Statistical Rate Monotonic Scheduling," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 123-132, 1998.

[11] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach," *Proceedings: IEEE Workshop on Real-Time Operating Systems*, 1992.

[12] Sara Baase, *Computer Algorithms*, Second Edition, Addison-Wesley Publishing Company, 1988.

[13] T. Bäck and F. Hoffmeister, "Extended Selection Mechanisms in Genetic Algorithms," In *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 92-99, 1991.

[14] Theodore P. Baker, "A Stack-Based Resource Allocation Policy for Realtime Processes," *The Real-Time Systems Journal,* vol 3, no 1, pp 67-100, 1991.

[15] Suman Banerjee and Ashok K. Agrawala, "Estimating Available Capacity of a Network Connection," in *Proceedings of the IEEE International Conference on Networks*, *Singapore*, 2000.

[16] Michael Barbanov and Victor Yodaikin, "Real-time Linux", *Linux Journal*, March 1996.

[17] Guillem Bernat, Alan Burns, and Albert Llamosí, "Weakly Hard Real-Time Systems," *IEEE Transactions on Computers*, vol. 50, no. 40, 2001, pp. 308-321.

[18] Guillem Bernat, Antoine Colin, and Stefan M. Petters, "WCET Analysis of Probabilistic Hard Real-Time System," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 279-288, 2002.

[19] Christian Bierwirth and Dirk C. Mattfeld, "Production Scheduling and Rescheduling with Genetic Algorithms," *Evolutionary Computation*, vol 7, no. 1, pp. 1-17, 1999.

[20] J. Blazewicz, M. Dror, and J. Welgarz, "Mathematical Formulations for Machine Scheduling: A Survey," *European Journal of Operations Research*, vol 51, no. 3, 1991, pp. 283-300.

[21] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, 2000.

[22] P. Bratley, M. Florian, and P. Robillard, Scheduling with Earliest Start and Due Date Constraints, *Naval Research Quarterly*, vol. 18, no. 4, 1971.

[23] R. Breyer and S. Riley, *Switched, Fast, and Gigabit Ethernet*, New Riders, Indianapolis, Indiana, 1999.

[24] Peter Brucker, *Scheduling Algorithms*, Springer-Verlag, Berlin, Germany, 1998.

[25] Peter Brucker, B. Jurish, and A. Krämer, "The Job-Shop Problem and Immediate Selection," *Annals of Operations Research*, vol 50, 1994, pp. 73-114.

[26] Giorgio C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Boston, Massachusetts, 1997.

[27] Erick Cantu-Paz, "A survey of Parallel Genetic Algorithms," *Calculators Pralleles*, Vol 10, no. 2, pp. 141-171, 1998.

[28] Todd Carpenter, Kevin Driscoll, Ken Hoyme, and Jim Carciofini, "ARINC 659 Scheduling: Problem Definition, " In *Proceedings of the Real-Time Systems Symposium*, 1994.

[29] Oliver Catoni, "Solving Scheduling Problems by Simulated Annealing," *SIAM Journal on Control and Optimization*, vol 36, no 5, pp. 1539-1575, 1998.

[30] Srigurunath Chakravarthi, *Predictability and Performance Factors Influencing the Design of Real-Time Messaging Layers*, MS Thesis, Department of Computer Science, Mississippi State University, 2000.

[31] Albert M. K. Cheng, *Real-Time Systems: Scheduling, Analysis, and Verification*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2002.

[32] Cheng-Chung Cheng and Steven. F. Smith, *Applying Constraint Satisfaction Techniques to Job Shop Scheduling (The Long Version)*, Technical Report CMU-RI-TR-95-03, Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

[33] Chih-Che Chou and Kang G. Shin, "Statistical Real-Time Channels on Multiaccess Bus Networks," *IEEE Transactions on Parallel and Distributed Systems*, Vol 8, No 8, pp. 769-780, 1997.

[34] Chih-Che Chou and Kang G. Shin, "Statistical Real-Time Video Channels over a Multiaccess Network," in *Proceedings of High-Speed Networking and Multimedia Computing*, pp 86-96, 1994.

[35] Lon-Chan Chu and Benjamin W. Wah, "Optimization in Real Time," *Proceedings: The 1991 IEEE Real Time Systems Symposium*, pp. 150-159, 1991.

[36] F. Della Croce, G. Menga, R. Tadei, M. Cavalotto, and L. Petri, "Cellular Control of Manufacturing Systems," *European Journal of Operations Research*, vol 69, pp. 498-509, 1993.

[37] Mark Crovella, Prakash Das, Czarek Dubnicki, Tomas LeBlanc, and Evangelos Markatos, "Multiprogramming on Multiprocessors," *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pp. 590-597, 1991.

[38] R. L. Cruz, "A Calculus for Network Delay, Part I: Network Elements in Isolation," *IEEE Transactions on Information Theory*, Vol 37, No. 1, pp. 114-131, 1991.

[39] R. L. Cruz, "A Calculus for Network Delay, Part II: Network Analysis," *IEEE Transactions on Information Theory*, Vol 37, No. 1, pp. 132-141, 1991.

[40] Zhenqian Cui, Quality of Service *Communication and Analysis of RSVP Applicability on Sub-Network*, MS Thesis, Department of Computer Science, Mississippi State University, 1999.

[41] Yoginder S. Dandass, "A Genetic Algorithm for Scheduling Directed Acyclic Graphs in the Presence of Communication Contention, " to appear in *Proceedings of the 17th Annual International Symposium on High Performance Computing Systems and Applications*, 2003.

[42] L. Davis, "Applying Adaptive Algorithms to Epistatic Domains," in *Proc. of the 9th International Joint Conference on Artificial Intelligence*, pp. 162-164, 1985.

[43] M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Information Processing*, vol 74, 1974.

[44] Rutvik Desai and Rajendra Patil, "SALO: Combining Simulated Annealing and Local Optimization for Efficient Global Optimization," *Proceedings: The 9th Florida AI Research Symposium (FLAIRS-'96)*, pp. 233-237, 1996.

[45] Jay L. Devore, *Probability & Statistics for Engineering and the Sciences*, Brooks/Cole Publishing Company, Monterey, California, 1982.

[46] José Luis Diaz, Daniel F. Garcia, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella, "Stochastic Analysis of Periodic Real-Time Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp 289-300, 2002.

[47] Stewart Edgar and Alan Burns, "Statistical Analysis of WCET for Scheduling," in *proceedings of the IEEE Real-Time Systems Symposium*, pp. 215-224, 2001.

[48] Hesham El-Rewini, Ted G. Lewis, and Hesham H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[49] Andreas Ermedahl, Hans Hansson, Mikael Sjődin, "Response-Time Guarantees in ATM Networks," in *Proceedings of the Real-Time Systems Symposium*, pp. 274-284, 1997.

[50] M. L. Fisher, "Optimal Solution of Scheduling Problems using Lagrange Multipliers: Part I," *Operations Research* vol 21, pp. 1114-1127, 1973.

[51] M. L. Fisher, "Optimal Solution of Scheduling Problems using Lagrange Multipliers: Part II," *Proceedings: Symposium on the Theory of Scheduling and its Applications*, Springer, Berlin, 1973.

[52] Eugene C. Freuder and Richard J. Wallace, "Partial Constraint Satisfaction," *Artificial Intelligence*, vol 58, no. 1-3, pp. 21-70, 1992.

[53] Dror G. Feitelson, *Job Scheduling in Multiprogrammed Parallel Systems (Extended Version)*, IBM Research Report RC 19790 (87657) Second Revision, http://citesseer.nj.nec.com/feitelson97job.html, 1997.

[54] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevick, and Parkson Wong, "Theory and Practice in Parallel Job Scheduling," *Job Scheduling Strategies for parallel Processing, Lecture Notes in Computer Science Volume 1291*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, pp. 1-34, 1997.

[55] Dror G. Feitelson and Larry Rudolph, "Parallel Job Scheduling: Issues and Approaches," *Job Scheduling Strategies for parallel Processing, Lecture Notes in Computer Science Volume 949*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, pp. 1-18, 1995.

[56] Dror G. Feitelson and Larry Rudolph, "Gang Scheduling Performance Benefits for Fine Grain Synchronization," *Journal of Parallel and Distributed Computing*, vol 14, no. 4, pp 306-318, 1992.

[57] Dror G. Feitelson and Morris A. Jette, "Improved Utilization and Responsiveness with Gang Scheduling," *Job Scheduling Strategies for parallel Processing, Lecture Notes in Computer Science Volume 1291*, Springer-Verlag, pp. 238--261, 1997.

[58] Domenico Ferrari, "A New Admission Control Method for Real-Time Communication in an Internetwork," *Advances in Real-Time Systems*, Sang H. Son (ed.), Chapter 5, pp. 105-116, 1995.

[59] Domenico Ferrari, "Real-Time Communications in an Internetwork," *Journal of High Speed Networks*, Vol 1, no 1, pp. 79-103, 1992.

[60] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm, "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," in *Proceedings of the Operating Systems Design and Implementation Symposium*, pp. 87-100, 1999.

[61] Mark K. Gardner, *Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems*, Ph.D. Thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1999.

[62] Kaushik Ghosh, Bodhisattwa Mukherjee, and Karsten Schwan, *A Survey of Real-Time Operating Systems*, Technical Report GIT-CC-93/18, Georgia Institute of Technology, Atlanta, Georgia, 1994.

[63] Steve Goddard and Kevin Jeffay, "The Synthesis of Real-Time Systems from Processing Graphs," In *Proceedings of the Fifth IEEE International Symposium on High Assurance Systems Engineering*, Albuquerque, New Mexico, pp. 177-186, 2000.

[64] V. S. Gordon and D. Whitley, "Serial and Parallel Genetic Algorithms as Function Optimizers," Technical Report CS-93-114, Colorado State University, 1993.

[65] Martin Grajcar, "Strengths and Weaknesses of Genetic List Scheduling for Heterogeneous Systems," in *Proceedings of the International Conference on Application of Concurrency to System Design*, pp. 123-132, 2001.

[66] Martin Grajcar, "Conditional Scheduling for Embedded Systems Using Genetic List Scheduling," in *Proceedings of the 13th International Symposium on System Synthesis*, pp. 123-128, 2000.

[67] Martin Grajcar, "Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System," in *Proceedings of the 36th Design Automation Conference*, pp. 280-285, 1999.

[68] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MPI Press, Cambridge, Massachusetts, 1994.

[69] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications," *Proceedings: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 1991.

[70] Hans Hansson, Andreas Ermedahl, K. W. Tindell, "Guaranteeing Real-Time Traffic Through an ATM Network," in *Proceedings of the 30th Hawaii International Conference on System Sciences, Volume 5*, pp. 44-53, 1997.

[71] Christopher A. Healey, David B. Whalley, and Marion G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp 288-297, 1995.

[72] John H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, Michigan, 1975.

[73] W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, vol 21, 1974

[74] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol 19, no. 6, pp.841-848, 1961.

[75] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, vol 18, no. 2, pp. 244–257, 1989.

[76] IEEE Standard 802.5-1989, *Token Ring Access Method and Physical Layer Specifications*, Institute of Electrical and Electronic Engineers, New York, 1989.

[77] Intel Corporation, *IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2002.

[78] J. R. Jackson, *Scheduling a Production Line to Minimize Maximum Tardiness*, Management Science Research Project 43, University of California, Los Angeles, 1955.

[79] Anant Singh Jain and Sheik Meeran, *A State-of-the-Art Review of Job-Shop Scheduling Techniques*, Technical Report, Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, Dundee, Scotland, 1998.

[80] Jan Jonsson and Kang G. Shin, "A Parameterized Branch-and-Bound Strategy for Scheduling Precedence-Constrained Tasks on a Multiprocessor System," *Proceedings: The International Conference on Parallel Processing (ICPP'97)*, pp. 158-165, 1997.

[81] K42 Team, "Scheduling in K42," white paper, http://www.research.ibm.com/K42/white-papers/Scheduling.pdf, modified on August 2002.

[82] Sanjay Kamat and Wei Zhao, "Real-Time Performance of Two Token Ring Protocols," *in Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Pennsylvania, pp. 347-354, 1993.

[83] Dong-In Kang; Richard Gerber, and Manas Saksena, "Parametric Design Synthesis of Distributed Embedded Systems," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1155–1169, 2000.

[84] Samuel Karlin and Howard M. Taylor, *A First Course in Stochastic Processes*, Academic Press, New York, 1975.

[85] J. B. Kim, T. Suda, and M. Yoshimura, "International Standardization of B-ISDN," *Computer Networks and ISDN Systems*, Vol 27, pp. 5-27, 1994.

[86] S. J. Kim and James C. Browne, "A General Approach to Mapping of Parallel Computation on Multiprocessor Architectures," In *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 1-8, 1988.

[87] B. Kruatrachue and Ted G. Lewis, "Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems," Technical Report, Oregon State University, Corvallis, OR, 1987.

[88] Seok-Kyu Kweon, Kang G. Shin, and Q. Zheng, "Statistical Real-Time Communication over Ethernet for Manufacturing Automation Systems," in *Proceedings IEEE Real-Time Technology and Applications Symposium*, 1999.

[89] Seok-Kyu Kweon, Kang G. Shin, and Gary Workman, "Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing," in *Proceedings IEEE Real-Time Technology and Applications Symposium*, 2000.

[90] Yu-Kwong Kwok and Ishfaq Ahmad, "Parallel Program Scheduling Techniques," Chapter 23 in *High Performance Cluster Computing Volume 1*, Rajkumar Buyya (ed.), Prentice Hall, Englewood Cliffs, New Jersey, pages 553–578, 1999.

[91] Yu-Kwong Kwok and Ishfaq Ahmad, "Static Scheduling for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol 31, no. 4, pp. 406-471, http://citeseer.nj.nec.com/314946, 1998.

[92] Yu-Kwong Kwok and Ishfaq Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors using a Parallel Genetic Algorithm," *Journal of Parallel and Distributed Computing*, vol 47, no. 1, pp. 58-77, 1997.

[93] Yu-Kwong Kwok and Ishfaq Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol 7, no. 5, pp 506-521, 1996.

[94] P. J. M. Van Laarhoven, E. H. L Aarts, and J. K. Lenstra, "Job Shop Scheduling by Simulated Annealing," *Operations Research*, vol 40, no 1, pp. 113-125, 1992.

[95] Gerardo Lamastra, Guiseppe Lipari, Giorgio Buttazzo. Antonio Casile, Fabio Conticelli, "HARTIK 3.0: a portable system for developing real-time applications," In *Proceedings of the Fourth International Workshop on Real-Time Computing Systems and Applications*, pp. 43-50, 1997.

[96] Eugene L. Lawler, "Recent Results in the Theory of Machine Scheduling," *Mathematical Programming: The State of the Art*, A. Becham et al. (eds), Springer-Verlag, New York, pp. 202-233, 1993.

[97] John P. Lehoczky, and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[98] John P. Lehoczky, Lui Sha, and Ye Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proceedings: IEEE Real-Time Systems Symposium*, pp. 166-171*,* 1989.

[99] John P. Lehoczky, Lui Sha, Jay K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," In *Proceedings of IEEE Real-Time Systems Symposium*, 1987.

[100] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," *Annals of Discrete Mathematics*, vol 1, pp. 343-362, 1977.

[101] J. Leung and J. W. Whitehead, "On The Complexity of Fixed Priority Scheduling of Periodic real-time Tasks," *Performance Evaluation,* vol 2, no. 4, 1982.

[102] Chengzhi Li, Riccardo Bettati, and Wei Zhao, "Static Priority Scheduling for ATM Networks," in *Proceedings of the Real-Time Systems Symposium*, pp. 264-273, 1997.

[103] Yau-Tsun Steven Li, Sharad Malik, Andrew Wolfe, "Efficient Microarchitecture Modeling and Pathway Analysis for Real-Time Software," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp 298-307, 1995.

[104] David A. Lifka, "The ANL/IBM SP Scheduling System," *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science Volume 949*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, pp. 295-303, 1995.

[105] Sung-Soo Lim; Jung Hee Han; Jihong Kim; Sang Lyul Min, "A Worst Case Timing Analysis Technique for Multiple-Issue Machines," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp334-345, 1998.

[106] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Choug Sang Kim, "An Accurate Worst Case Timing Analysis Technique for RISC Processors," in *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pp. 97-108, 1994.

[107] Shyh-Chang Lin, Erik D. Goodman William F. Punch, III, "Investigating Parallel Genetic Algorithms on Job Shop Scheduling Problems," *Evolutionary Programming VI: Proceedings of the 6th Annual Conference on Evolutionary Programming*, Peter J. Angeline *et al*. (eds.), Springer-Verlag, Lecture Notes in Computer Science Vol 1213, pp. 383-394, 1997.

[108] Jinfeng Liu, Pai H. Chou, Nader Bagherzadeh, and Fadi Kurdahi, "Power-Aware Scheduling under Timing Constraints for Mission-Critical Embedded Systems," in *Proceedings of the 38th Design Automation Conference*, pp. 840-845, 2001.

[109] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal for the Association of Computing Machinery* vol 20, no. 1, pp. 46-61, 1973.

[110] T. Lundqvist and P. Stenström, "An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution," *Real-Time Systems*, vol. 17 no. 2-3, pp. 183-207, 1999.

[111] Lynux Works, Inc. *LynuxWorks Patented Technology Speeds Handling of Hardware Events: Meeting real-time performance requirements*, white paper, http://www.lynuxworks.com/products/whitepapers/patentedio.php3, 2002.

[112] Nicholas Malcolm and Wei Zhao, "Hard Real-Time Communication in Multiple-Access Networks, *Real-Time Systems*, Vol 9, pp. 79-107, 1995.

[113] Alan S. Manne, "On the Job-Shop Scheduling Problem," *Operations Research*, vol 8, no 2, 1960.

[114] G. B. McMahon and M. Florian, "On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness," *Operations Research*, vol 23, no. 3, pp. 475-482, 1975.

[115] Zbigniew Michalewicz, *Genetic algorithms + data structures = evolution programs*, Springer-Verlag, New York, 1992.

[116] Steven Minton, Mark D. Johnston, Adrew B. Phillips, Phillip Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems," *Artificial Intelligence*, vol 58, no. 1-3, pp. 161-205, 1992.

[117] Melanie Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, Massachusetts, 1996.

[118] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa, *Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel*, IEEE Workshop on Real-Time Operating Systems and Software, May 1990.

[119] Yannick Monnier, Jean-Pierre Beauvais, and Anne-Marie Déplanche, "A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System," in *Proceedings of the 24th EUROMICRO Conference*, vol II, pp. 708-714, 1998.

[120] Myricom, Inc., *Myrinet Overview*, http://www.myrinet.com/myrinet/overview/index.html, 2002

[121] Marco Di Natale and John A. Stankovic, "Scheduling Distributed Real-Time Tasks with Minimum Jitter," *IEEE Transactions on Computers*, vol. 49, no. 4, pp. 303-316, 2000.

[122] Joseph Kee-Yin Ng, Shibin Song, and Wei Zhao, "Integrated Delay Analysis of Regulated ATM Switch," in *Proceedings of the Real-Time Systems Symposium*, pp. 285-296, 1997.

[123] Douglass Niehaus, William Dinkel, and Sean B. House, "Effective real-time system implementation with KURT Linux," In *Proceedings of the Real-Time Linux Workshop*, 1999.

[124] Douglas Niehaus, Erich M. Nahum, John A. Stankovic, Kirthi Ramamritham, *Architecture and OS Support for Predictable Real-Time Systems*, Spring internal document, http://www-ccs.cs.umass.edu/spring/internal/arch_os_support.ps, March 1992.

[125] Walter Oney, *Programming the Microsoft Windows Driver Model*, Microsoft Press, Redmond, Washington, 1999.

[126] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proceedings: 3rd International on Distributed Computing Systems*, pp. 22-30, 1982.

[127] J. C. Potts, T. D. Giddens, and S. B. Yadav, "The Development and Evolution of an Improved Genetic Algorithm Based on Migration and Artificial Selection," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 1, pp. 73-86, 1994.

[128] Liam B. Quinn and Richard G. Russell, *Fast Ethernet*, John Wiley & Sons, Inc., New York, 1997.

[129] Minsoo Ryu and Seung-Jean Kim, "Deterministic and Statistical Deadline Guarantees for a Mixed Set of Periodic and Aperiodic Tasks," in *Proceedings of the International Conference on Real-Time and Embedded Computing Systems and Applications*, pp 232-247, 2003.

[130] N. Sadeh and Y. Nakakuki, "Focused Simulated Annealing Search – An Application to Job-Shop Scheduling," *Annals of Operations Research*, vol 63, pp. 77-103, 1996.

[131] Manas Saksena, James da Silva and Ashok K. Agrawala, "Design and Implementation of Maruti-II," *Advances in Real-Time Systems*, Sang H. Son (ed.), Chapter 4, pp. 72-101, 1995.

[132] Manas Saksena, *Parametric Scheduling for Hard Real-Time Systems*, Ph.D. Dissertation, Department of Computer Science, University of Maryland, 1994.

[133] F. E. Sandnes and Graham M. Megson, "Improved Static Multiprocessor Scheduling using Cyclic Task Graphs: A Genetic Approach," in *Proceedings of Parallel Computing* (*PARCO'97*), pp. 703-710, 1997.

[134] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.

[135] R. Sethi, "Scheduling Graphs on Two Processors," *SIAM Journal of Computing*, vol 5, no 1, pp73-82, 1976.

[136] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Parallel and Distributed Computing*, vol. 39, no. 3, 1990.

[137] N. V. Shakhlevich, Y. N. Sotskov, K. Krueger, F. Werner, "A Decomposition Algorithm for Scheduling Problems on Mixed Graphs," *Journal of the Operational Research Society*, vol 46, pp. 1481-1497, 1995.

[138] David B. Shmoys, Clifford Stein, and Joel Wein, "Improved Approximation Algorithms for Shop Scheduling Problems," *SIAM Journal of Computing* vol 23, pp. 617-632, 1994.

[139] Gilbert C. Sih and Edward. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol 4, no. 2, pp 175-187, 1993.

[140] Anthony Skjellum, Arkady Kanevsky, Yoginder S. Dandass, Jerrell Watts, Steve Paavola, Dennis Cottel, Greg Henley, L. Shane Hebert, Zhenqian Cui, Anna Rounbehler, and The Real-Time Message Passing Interface Forum, "The Real-Time Message Passing Interface," *Concurrency and Computation: Practice and Experience*, accepted, in press, 2003.

[141] Patrick G. Sobalvarro and William E. Weihl. "Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors," *Job Scheduling Strategies for parallel Processing, Lecture Notes in Computer Science Volume 949*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995.

[142] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien, "Dynamic Coscheduling on Workstation Clusters," *Proceedings of the International Parallel Processing Symposium* (IPPS'98), 1998.

[143] Edward Solari and George Willse, *PCI Hardware and Software*, Annabooks, San Diego, California, 1998.

[144] David A. Solomon and Mark E. Russinovich, *Inside Microsoft Windows 2000, Third Edition*, Microsoft Press, Redmond, Washington, 2000.

[145] Marco Spuri and Giorgio C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling," in *Proceedings IEEE Real-Time Systems Symposium*, pp. 12-21, 1994.

[146] William Stallings, *Operating Systems: Internals and Design Principals*, 3rd Edition, Prentice Hall, New Jersey, 1997.

[147] John A. Stankovic, "Misconceptions About Real-Time Computing," *IEEE Computer*, pp. 10-19, Oct. 1988.

[148] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese, *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, The MIT Press, Cambridge, Massachusetts, 1999.

[149] James K. Strayer, *Linear Programming and its Applications*, Springer-Verlag, New York, 1989.

[150] Jay K. Strosnider and Thomas E. Marchok, "Responsive, Deterministic IEEE 802.5 Token Ring Scheduling," *Real-Time Systems*, Vol 1, No 2, pp. 133-158, 1989.

[151] Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao, "Real-Time Mach: Towards a Predictable Real-Time System," *Proceedings of USENIX Mach Workshop*, October 1990.

[152] T. S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L. C. Wu, and J. W. S. Liu, "Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 164-173, May 1995.

[153] J. Ullman, "NP-Complete Scheduling problems," *Journal of Computer and System Sciences*, vol 10, pp. 384-393, 1975.

[154] Chitra Venkatramani and Tzi-cker Chiueh, "Design, Implementation, and Evaluation of a Software-based Real-Time Ethernet Protocol," *Computer Communication Review*, Vol. 24, No. 4, 1995.

[155] Wind River Systems, Inc. *pSOSystem 3 Datasheet*, http://www.windriver.com/products/psosystem_3/psosystem_3.pdf, 2002.

[156] Wind River Systems, Inc. *VxWorks 5.x Datasheet*, http://www.windriver.com/products/vxworks5/vxworks_54.pdf, 2002.

[157] Min-You Wu and Daniel D. Gajski, "Hypertool: A Programming Aid for Message Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol 1, no. 3, pp. 330–343, 1990.

[158] Jia Xu and David L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, vol 16, no. 3, pp. 360-369, 1990.

[159] Dong Xuan, Chengzhi Li, Riccardo Bettati, Jianer Chen, and Wei Zhao, "Utilization-Based Admission Control for Real-Time Applications," in *Proceedings of the 2000 International Conference on Parallel Processing*, pp. 251-260, 2000.

[160] Tao Yang and Apostolos Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol 5, no. 9, pp. 951-967, 1994.

[161] Tao Yang and Apostolos Gerasoulis, "On the Granularity and Clustering of Directed Acyclic Task Graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol 4, no. 6, pp 686-701, 1993

[162] Marat Zhaksilikov and Frederick Harris, Jr., "Comparison of Different Implementations of Parallelization of Genetic Algorithms," *Proceedings: ISCA's International Conference on Intelligent Systems*, Reno, NV, 1996.

[163] Hui Zhang and Domenico Ferrari, "Rate-Controlled Service Disciplines," *Journal of High-Speed Networks*, vol. 3, no. 4, pp. 389-412, 1994.

[164] Hui Zhang and Domenico Ferrari, "Rate-Controlled Static-Priority Queuing," in *Proceedings of IEEE INFOCOM'93 Conference on Computer Communications*, 1993.

[165] Hui Zhang and Edward W. Knightly, "A New Approach to Support Delay-Sensitive VBR Video in Packet-Switched Networks," in *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 275-286, 1995.

[166] L. Zhang, R. Braden, D. Estrin, S. Herzog, and S. Jamin, *Resource ReSerVation Protocol (RSVP)*, RFC, 1995.

[167] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network Magazine*, September 1993.

APPENDIX A

SUMMARY OF ADDITIONAL GENETIC LIST SCHEDULING EXPERIMENTS

This appendix presents results of schedule construction for the large size (with approximately 500 tasks) DAGs for all structure types using the stochastic GLS approach.

A.1 Comparison of Stochastic LS and Stochastic GLS

Table A.1 compares the performance of the GLS and LS approaches for the large HFJ DAGs. GLS produced shorter schedules for 10 of the 15 DAGs. The schedule improvement of GLS compared with LS averaged over all large HFJ DAGs is nearly 4%.

Table A.1 Comparison of GLS and LS Schedules for Large HFJ DAGs

Distribution Type	CCR	Length of GLS Schedule	Length of Best LS Schedule	Improvement in Schedule Length
Beta	0.5	47107	54702	13.88%
	0.67	40848	49621	17.68%
	1.0	35884	37844	5.18%
	1.5	46406	46502	0.21%
	2.0	56581	51739	-9.36%
Exponential	0.5	45369	54723	17.09%
	0.67	43097	49472	12.89%
	1.0	35047	37314	6.08%
	1.5	47635	45472	-4.76%
	2.0	58111	49808	-16.67%
Random	0.5	43922	51627	14.92%
	0.67	38439	43900	12.44%
	1.0	32624	36173	9.81%
	1.5	44217	42752	-3.43%
	2.0	56350	48132	-17.07%
			Average Improvement:	3.93%

Table A.2Comparison of GLS and LS Schedules for Large MVA DAGs

Distribution Type	CCR	Length of GLS Schedule	Length of Best LS Schedule	Improvement in Schedule Length
Beta	0.5	33377	43929	24.02%
	0.67	31483	34882	9.74%
	1.0	26089	29296	10.95%
	1.5	33332	35307	5.59%
	2.0	40446	40548	0.25%
Exponential	0.5	35488	42856	17.19%
	0.67	32715	34997	6.52%
	1.0	26606	30745	13.46%
	1.5	33268	35324	5.82%
	2.0	41409	41453	0.11%
Random	0.5	34723	39577	12.26%
	0.67	31013	34056	8.94%
	1.0	25524	28064	9.05%
	1.5	31740	34203	7.20%
	2.0	39976	39976	0.00%
			Average Improvement:	8.74%

Table A.2 compares the performance of the GLS and LS approaches for the large MVA DAGs. GLS produced shorter schedules for all 15 DAGs. The schedule improvement of GLS compared with LS averaged over all large MVA DAGs is nearly 8.74%.

Table A.3 Comparison of GLS and LS Schedules for Large OUT DAGs

Distribution Type	CCR	Length of GLS Schedule	Length of Best LS Schedule	Improvement in Schedule Length
Beta	0.5	47107	54702	13.88%
	0.67	40848	49621	17.68%
	1.0	35884	37844	5.18%
	1.5	46406	46502	0.21%
	2.0	56581	51739	-9.36%
Exponential	0.5	45369	54723	17.09%
	0.67	43097	49472	12.89%
	1.0	35047	37314	6.08%
	1.5	47635	45472	-4.76%
	2.0	58111	49808	-16.67%
Random	0.5	43922	51627	14.92%
	0.67	38439	43900	12.44%
	1.0	32624	36173	9.81%
	1.5	44217	42752	-3.43%
	2.0	56350	48132	-17.07%
			Average Improvement:	14.95%

Table A.3 compares the performance of the GLS and LS approaches for the large OUT DAGs. GLS produced shorter schedules for 10 of the 15 DAGs. The schedule improvement of GLS compared with LS averaged over all large OUT DAGs is nearly 15%.

Table A.4 Comparison of GLS and LS Schedules for Large RND DAGs

Distribution Type	CCR	Length of GLS Schedule	Length of Best LS Schedule	Improvement in Schedule Length
Beta	0.5	29673	41934	29.24%
	0.67	20680	30028	31.13%
	1.0	15810	20421	22.58%
	1.5	16803	20612	18.48%
	2.0	17254	19653	12.21%
Exponential	0.5	34475	54162	36.35%
	0.67	23502	33903	30.68%
	1.0	17656	21917	19.44%
	1.5	14319	15841	9.61%
	2.0	19986	24751	19.25%
Random	0.5	24591	29201	15.79%
	0.67	22816	29923	23.75%
	1.0	16165	19960	19.01%
	1.5	15179	19506	22.18%
	2.0	17188	20407	15.77%
			Average Improvement:	21.70%

Table A.4 compares the performance of the GLS and LS approaches for the large RND DAGs. GLS produced shorter schedules for all 15 DAGs. The schedule improvement of GLS compared with LS averaged over all large RND DAGs is 21.7%.

Table A.5 Comparison of GLS and LS Schedules for Large OUT DAGs

Distribution Type	CCR	Length of GLS Schedule	Length of Best LS Schedule	Improvement in Schedule Length
Beta	0.5	71426	79707	10.39%
	0.67	65339	71193	8.22%
	1.0	57943	64710	10.46%
	1.5	72611	75452	3.77%
	2.0	120268	91659	-31.21%
Exponential	0.5	72339	80488	10.12%
	0.67	67275	74021	9.11%
	1.0	55913	62648	10.75%
	1.5	76124	79219	3.91%
	2.0	101908	89210	-14.23%
Random	0.5	70268	77523	9.36%
	0.67	67517	75760	10.88%
	1.0	55375	60660	8.71%
	1.5	74086	77899	4.89%
	2.0	99346	87963	-12.94%
			Average Improvement:	2.81%

Table A.5 compares the performance of the GLS and LS approaches for the large SFJ DAGs. GLS produced shorter schedules for 3 of the 15 DAGs. The schedule improvement of GLS compared with LS averaged over all large SFJ DAGs is 2.81%.

These tables show that, in general, the stochastic GLS approach produces shorter schedules than the stochastic LS approach. However, for many cases, the stochastic Ls approach was better. Because the execution time of the GLS approach is significantly greater than that of the LS approach, it is evident that both approaches be utilized in order to find a high-quality schedule.

A.2 QoS-Performance Tradeoff with GLS

Figure A.1 plots the schedule compression achieved by using the GLS approach for all DAGs grouped by structure type. This chart shows that significant schedule compression can be achieved when the GLS approach is used to construct schedules.

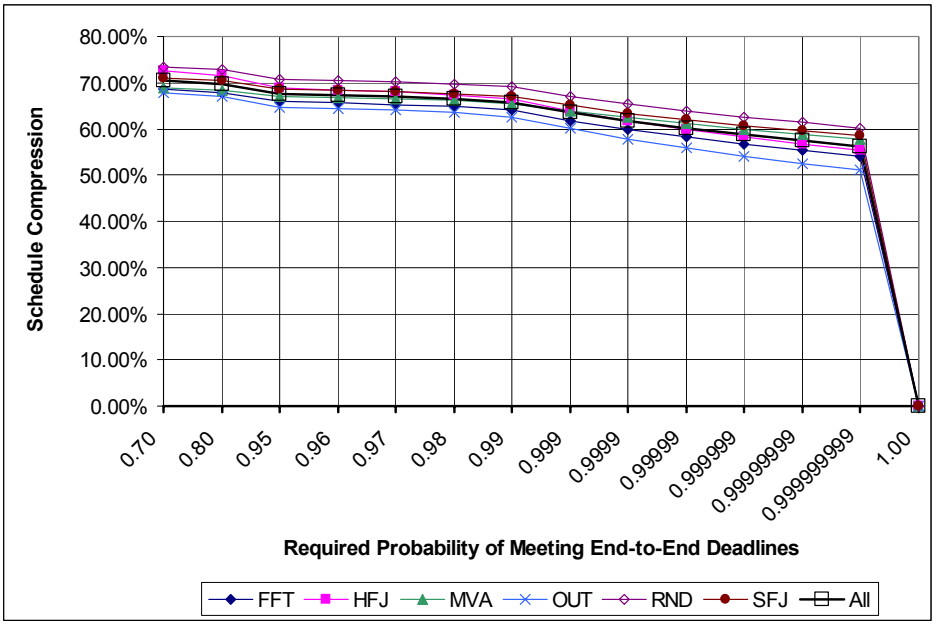


Figure A.1 GLS Schedule Compression Grouped by Structure

Figure A.2 plots the QoS-performance tradeoff metric for the schedules produced by the GLS approach for all DAGs grouped by structure type. This chart supports the idea that reducing the required probability of meeting end-to-end deadlines relative to the WCET requirements can result in significant dividends in terms of reduced schedule lengths.

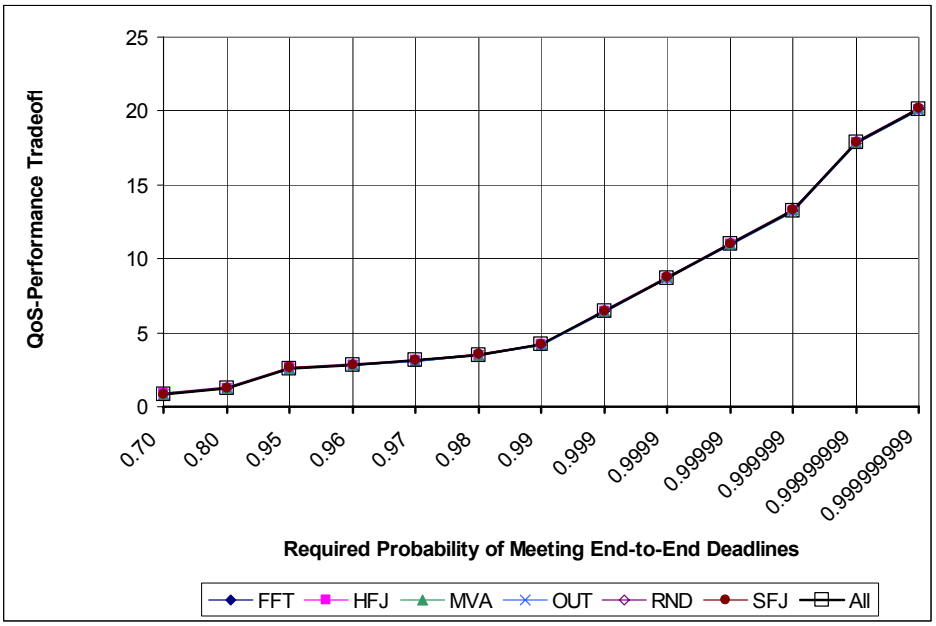


Figure A.2 GLS QoS-Performance Tradeoff Grouped by Structure

A.3 Jitter Control with GLS

Figure A.3 plots the average stochastic jitter factor grouped by structure type resulting from specifying various amounts of jitter control to the GLS algorithm for all large DAGs. Schedules for the SFJ DAGs have the larges amount of inherent jitter. However, an application of 5% jitter control significantly reduces the jitter in the SFJ DAGs. Figure A.4 plots the stochastic utilization of resources in the schedules produced for all large DAGs by the GLS algorithm. This chart shows that increasing jitter control improves utilization.

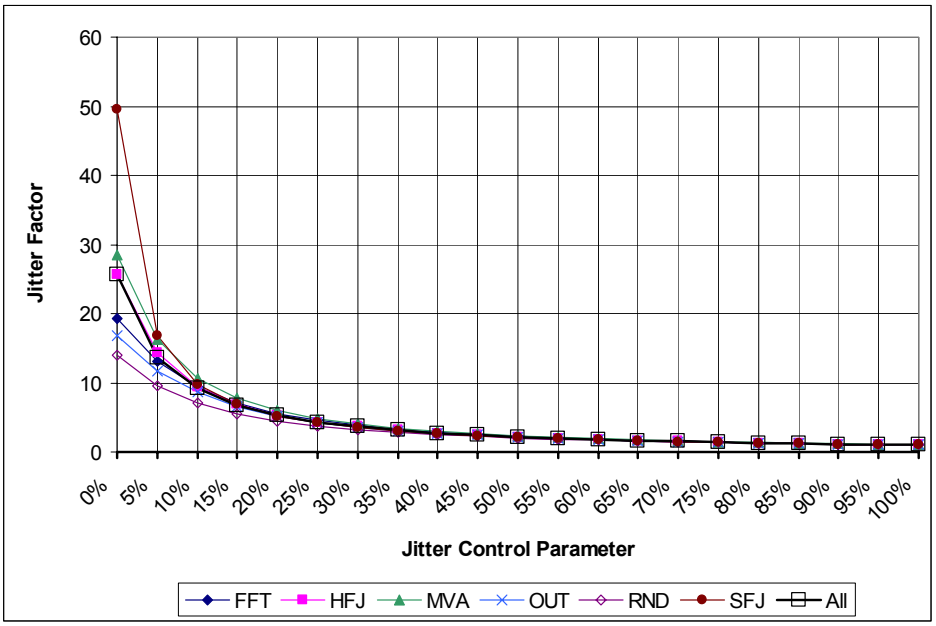


Figure A.3 GLS Jitter Control Grouped by Structure

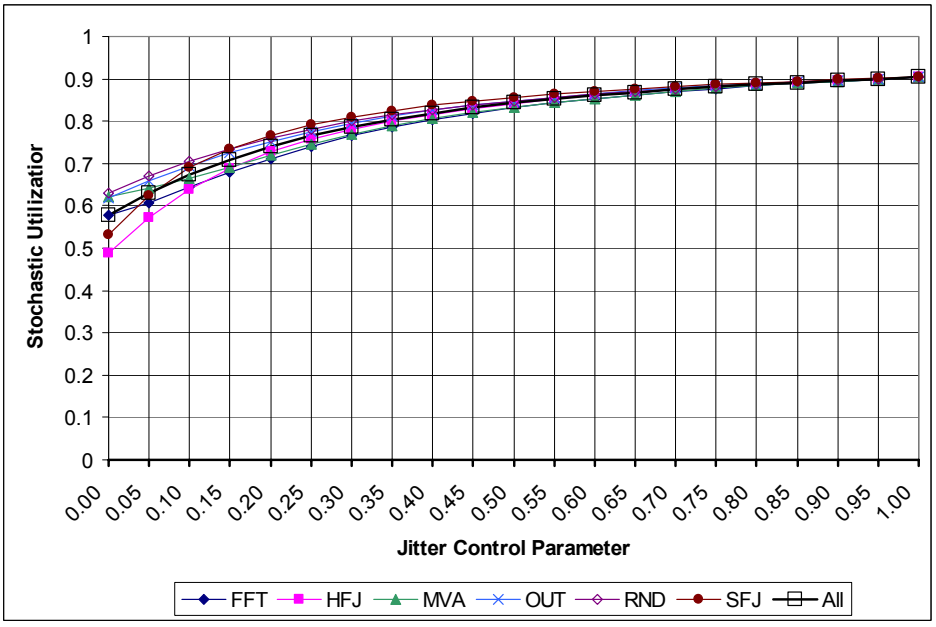


Figure A.4 GLS Utilization Grouped by Structure

A.4 Trading-off Performance for QoS with GLS

Figure A.5 plots the schedule compression metric in response to changes in the required probabilities for meeting end-to-end deadlines for specific jitter control parameter values averaged over all large DAGs when using the GLS approach. The figure shows that the maximum compression increases when moderate amount of jitter control is applied. However, compression decreases when the jitter control parameter is increased to values above 0.50. This result is similar to the result in Figure 5.51 showing the compression in response to the jitter control parameter for LS algorithms.

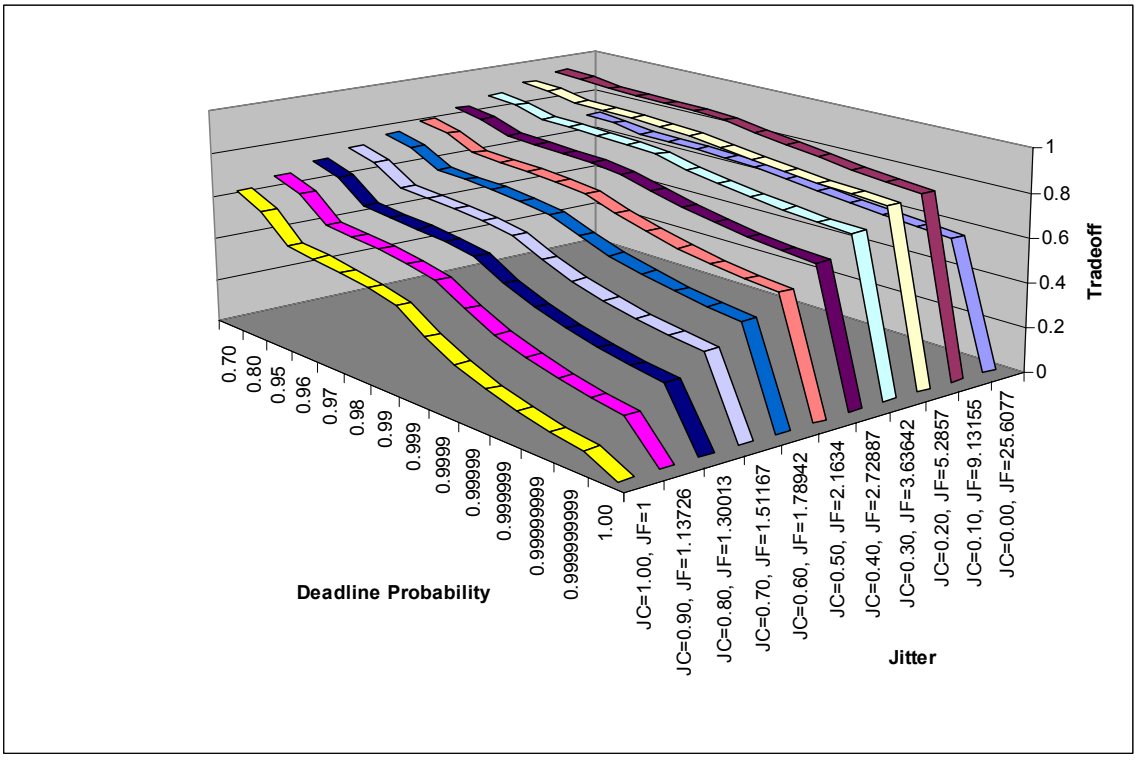


Figure A.5 GLS Compression vs. Jitter Control Factor

The results in this appendix show that GLS approach, when applied to large DAGs with a variety of structures, produces schedules with characteristics that are consistent with the schedules produced for the FFT DAGs.